National Technical University of Athens
School of Electrical and Computer Engineering
Operating Systems Lab
Winter Semester 2022-2023

Team oslab30
Ioannis Dressos - 03119608

Report of 1st Lab Exercise (Riddle)

We perform a simple behavioral analysis and runtime manipulation of the given executable on a Debian Linux x64 VM with the use of operating system utilities like **strace**.

For the completion of the exercise we will often use small programs written in the Python 3 programming language, because of the seamless access it provides to Unix system calls.

Challenge 0

The program performs the openat() system call (without the O_CREAT flag) in an attempt to open the file .hello_there, which however does not exist. We create the file and re-run the program:

```
touch .hello_there
```

Challenge 1

The program attempts to write() to the .hello_there file, and the challenge fails if the attempt is successful. We revoke the file's write privileges from the user and re-run the program:

```
chmod u-w .hello_there
```

Challenge 2

The program performs the alarm() system call to schedule a SIGALRM signal to be sent to the process after a couple of seconds.
Afterwards, pause() is called to halt the process.

We must send **SIGCONT** to the process through kill or htop before SIGALRM:

```
kill -s SIGCONT {PID}
```

We can do the same by pressing **CTRL+Z** to halt the process and then running **fg** to continue its execution.

Challenge 3

This time we cannot see any syscall activity through strace, so we try debugging with **ltrace**.

The program uses **getenv()** to read the ANSWER environment variable. We must look up the correct answer to the challenge's question "what is the answer to life, the universe, and everything?", set the variable and re-run the program:

```
export ANSWER=42
```

Challenge 4

Back to using **strace**, the program attempts to open the file magic_mirror, which does not exist. We create the file and re-run the program:

```
touch magic_mirror
```

Now we observe that the program writes a random character to the file, then attempts to read() a single character from the file. It repeats this process multiple times until the challenge fails.

Assuming that the program wants to read the same character it writes to the file every time, we decide that a FIFO pipe is the way to go (which is obviously hinted anyway):

```
mkfifo magic_mirror
```

Challenge 5

The program uses the **fcntl()** system call with the F_GETFD flag to read the flags of file descriptor 99. That file descriptor however does not exist.

The only way to create a single specific file descriptor is by duplicating an already existent one with the **dup2()** system call.

We must write a program that does that and launches riddle as a child process so it inherits the file descriptor table:

```
import os

print("Creating file descriptor..")
os.dup2(1, 99);

print("Executing riddle..")
pid = os.fork()
if pid == 0:
     os.execve("./riddle", ["./riddle"], os.environ.copy())
     exit()
os.waitpid(pid, 0)

print("\nClosing file descriptor..")
os.close(99);

print("Done!")
```

Alternatively, we can inject a false return value of 0 to all fcntl() system calls made by the program through strace:

```
strace -e inject=fcntl:retval=0 ./riddle
```

Challenge 6

If we run the program through **strace -f** we can see that it forks two child processes.

It is clear in the strace and program output that the child processes are trying to communicate with each other through file descriptors 33, 34, 53 and 54.

Same as before, we write a program that assigns the proper file descriptors to **Unix pipes** to allow for inter-process communication. Our program launches riddle as a child process so that it inherits the file descriptor table:

```
import os

print("Creating pipes..")

r1, w1 = os.pipe()
r2, w2 = os.pipe()

print("Duplicating file descriptors..")

os.dup2(r1, 33)
os.dup2(w1, 34)

os.dup2(r2, 53)
os.dup2(w2, 54)

print("Executing riddle..")
pid = os.fork()
if pid == 0:
     os.execve("./riddle", ["./riddle"], os.environ.copy())
     exit()
waitpid(pid, 0)

print("\nClosing file descriptors..")

for fd in [r1, w1, r2, w2, 33, 34, 53, 54]:
     os.close(fd)

print("Done!")
```

Challenge 7

The program uses the **lstat()** system call to read the status of files hey_there and .hello_there. The file .hey_there does not exist, so we create it. However the challenge still fails.

If we interpret the challenge hint correctly it becomes clear that we must create .hey_there as a **hard link** to .hello_there:

```
ln .hey_there .hello_there
```

The challenge succeeds.

Challenge 8

The program uses the openat(), **lseek()** and read() system calls to read byte 1073741824 of file bf00. It seems that the character it wants to read is the character 'X'.

We cannot create or modify a file in such a way using a text editor due to memory constraints. We must write a program for the job.

After completing the task it becomes apparent that the program performs the same check for all files from bf00 through bf09, so we modify our code accordingly:

```
import os

for i in range(10):
      file = "bf0" + str(i)
      try:
            fd = os.open(file, os.O_RDWR | os.O_CREAT)
            os.lseek(fd, 1073741824, os.SEEK_SET)
            os.write(fd, str.encode("X"))
            os.close(fd)

            print(file + ": Success")
      except (OSError, IOError):
            print("Could not open file: " + file)

print("Done!")
```

Challenge 9

It seems that the program is trying to **connect()** to a **socket()** on port 49842. At first, we use **netcat** to monitor any information sent by riddle through the socket:

```
nc -l -p 49842
```

The program connects to the socket and sends through it a random mathematical expression in the form of a question, then expects a response with the correct result.

To automate the whole process of passing this challenge, we write a program that among everything else also isolates the mathematical expression from the question, and evaluates it:

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
	s.bind(("127.0.0.1", 49842))
	s.listen()

	conn, addr = s.accept()

	with conn:
		print(f"{addr} connected")

		while True:
			data = conn.recv(1024)
			if not data:
				print("Connection closed")
				break

			q = data.decode("UTF-8")
			expr = q[12:len(q) - 2]

			try:
				conn.sendall(str.encode(str(eval(expr))))
				print("Challenge completed!")
			except SyntaxError:
				print("Could not evaluate expression: " + expr)
```

Challenges 10 & 11

The program creates the file secret_number, then memory maps it with the **mmap()** system call. Afterwards, the **unlink()** system call is used to delete the filename from the file system, but not the file itself.

→ According to Linux man pages, the file itself will be deleted when the last file descriptor pointing to it in a process is closed.

After unlinking the file, the program writes to its file descriptor the answer to a question it's asking us.

Since the file was memory-mapped before being unlinked, we can find its contents in the processes map files. To find the specific map file we refer to the return value of the mmap() system call which is visible in strace.

```
screen -dmS riddle strace ./riddle
ps all
cd /proc/{PID}/map_files
ls
cat {MAP_FILE}
```

The map file in question will most likely be colored red in **ls**.

Challenge 12

The program creates a temp file inside the /tmp directory. Part of the file's name is randomly generated.

We are given a few seconds to write a specific character that the program gives us at byte 111 of the temporary file.

We must write a program that reads the character as a command line argument, then automatically opens the correct temporary file and writes the character to it:

```
import os
import sys

if len(sys.argv) != 2:
	print("Usage: python3 challenge12.py <CHAR>")
	exit()

tempfiles = []

for(dirpath, dirnames, filenames) in os.walk("/tmp"):
	tempfiles = list(filter(lambda f: f.startswith("riddle-"),
filenames))
	break

if len(tempfiles) == 1:
	file = "/tmp/" + tempfiles[0]

	try:
		fd = os.open(file, os.O_WRONLY)

		os.lseek(fd, 111, os.SEEK_SET)
		os.write(fd, str.encode(argv[1]))
		os.close(fd)

		print("Success!")
	except (OSError, IOError):
		print("Could not open file: " + file)

elif len(tempfiles) == 0:
	print("No temp riddle files found")
	exit()
else:
	print("Found multiple temp riddle files - please clean up the
/tmp directory")
	exit()
```

Challenge 13

At first the program attempts to open the file .hello_there with the O_RDWR flag. The attempt fails because we revoked the user's write privileges for the file for challenge 2. We restore those privileges:

```
chmod u+w .hello_there
```

We re-run the program using strace. This time it looks like the program opens the file and uses the **ftruncate()** system call to increase its size to 32768 bytes.

Then, it memory-maps the file and uses ftruncate() again, this time to decrease its size. Finally it reads from stdin to write to the file.

Any attempt to write to the file will obviously raise **SIGBUS**. We must change its size back to 32768 bytes in order to avoid the error:

```
import os

file = ".hello_there"

try:
      fd = os.open(file, os.O_RDWR)

      os.ftruncate(fd, 32768)
      os.close(fd)

      print("Successfully truncated file: " + file)
except (OSError, IOError):
      print("Could not open file: " + file)
```

Challenge 14

The program seemingly uses the **getpid()** system call to check if it specifically has the PID 32767. If it doesn't, the challenge fails.

The operating system assigns PIDs to processes by reading the last assigned PID from the file **/proc/sys/kernel/ns_last_pid**.
→ If we set this file's content right before the riddle process spawns, then we can manipulate its PID.
However, we have to be careful in our implementation: another process can be forked and thus change the ns_last_pid file before riddle gets a chance to execute.

To solve this problem we can use the **flock()** system call, which allows us to acquire an exclusive lock on a file for the calling proccess only.

Before we jump into the code, it is worth mentioning that this challenge can be easily passed by manipulating the return value of the getpid() with strace like we did in challenge 5:

```
strace -e inject=getpid:retval=32767 ./riddle
```

When writing to the ns_last_pid file we will specify PID 32766 instead of 32767 because the file contains the PID that was last assigned to a process. When a new process is forked, it is assigned the PID that is the one in the file, incremented by 1.

```
import os, fcntl

if os.geteuid() != 0:
      print("Program requires root privileges")
      exit()

file = "/proc/sys/kernel/ns_last_pid"
try:
      fd = os.open(file, os.O_RDWR | os.O_CREAT)

      fcntl.flock(fd, fcntl.LOCK_EX)
      print("Acquired exclusive lock on file: " + file)

      print("Manipulating riddle execution PID..")
      os.write(fd, str.encode("32766"))

      print("Executing riddle..")
      pid = os.fork()
      if pid == 0:
            os.execve("./riddle", ["./riddle"], os.environ.copy())
            exit()
      os.waitpid(pid, 0)

      print("\nReleasing exclusive lock on file: " + file)
      fcntl.flock(fd, fcntl.LOCK_UN)

      os.close(fd)
      print("Done!")
except (OSError, IOError):
      print("An unexpected error occured")
```