Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλ. Μηχανικών & Μηχανικών Υπολογιστών
Εργαστήριο Λειτουργικών Συστημάτων
Χειμερινό Εξάμηνο 2022-2023

Ομάδα oslab30
Ιωάννης Δρέσσος - 03119608

Αναφορά 1ης Εργαστηριακής Άσκησης (Riddle)

Η ανάλυση της συμπεριφοράς του εκτελέσιμου διενεργείται μέσω Debian Linux x64 VM και με την χρήση εργαλείων του λειτουργικού συστήματος όπως το **strace**.

Τα προγράμματα που χρησιμοποιούνται για την επίλυση μερικών από τις προκλήσεις είναι γραμμένα σε Python 3 διότι προσφέρει εύκολη πρόσβαση σε κλήσεις συστήματος.

Πρόκληση 0

Το πρόγραμμα προσπαθεί να ανοίξει το αρχείο .hello_there το οποίο δεν υπάρχει με την κλήση συστήματος openat(). Το δημιουργούμε και τρέχουμε το πρόγραμμα ξανά:

```
touch .hello_there
```

Πρόκληση 1

Το πρόγραμμα ελέγχει αν μπορεί να γράψει στο αρχείο .hello_there με την κλήση συστήματος write(). Αν μπορεί, τοτε η πρόκληση αποτυγχάνει:

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

Πρόκληση 2

Το πρόγραμμα εκτελεί την κλήση συστήματος pause() για να διακόψει την διεργασία ενώ απο πρίν προγραμματίζει να σταλεί σήμα SIGALRM στην διεργασία μετά από λίγα δευτερόλεπτα με την κλήση συστήματος alarm().

Στέλνουμε σήμα **SIGCONT** στην διεργασία μέσω της εντολής kill η του προγράμματος htop:

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

Το ίδιο αποτέλεσμα μπορούμε να πετύχουμε διακόπτοντας την διεργασία με **CTRL+Z** και εκτελώντας **fg** για να την συνεχίσουμε.

Πρόκληση 3

Στην συγκεκριμένη πρόκληση δεν φαίνεται να εκτελείται κάποια κλήση συστήματος μέσω της strace, επομένως δοκιμάζουμε να τρέξουμε το εκτελέσιμο μέσω **ltrace**.

Το πρόγραμμα καλεί την **getenv()** για την μεταβλητή περιβάλλοντος ANSWER. Έπειτα από μια αναζήτηση για την σωστή απάντηση, ορίζουμε την μεταβλητή περιβάλλοντος και τρέχουμε ξανά εκτελέσιμο:

```
export ANSWER=42
```

Πρόκληση 4

Το εκτελέσιμο ανοίγει το αρχείο magic_mirror και γράφει σε αυτό έναν τυχαίο χαρακτήρα. Έπειτα διαβάζει από το ίδιο αρχείο έναν χαρακτήρα.

→ Αν δημιουργήσουμε το αρχείο magic_mirror, φαίνεται ότι επαναλαμβάνεται η διαδικασία που περιγράφεται παραπάνω πολλές φορές, κάθε φορά με διαφορετικό χαρακτήρα.

Καταλαβαίνουμε ότι το πρόγραμμα θέλει να διαβάσει τον ίδιο χαρακτήρα που γράφει κάθε φορά. Για την συγκεκριμένη δουλειά βολεύει να χρησιμοποιήσουμε ένα **Unix FIFO pipe**, όπως προδίδει και το πρόγραμμα:

```
mkfifo magic_mirror
```

Πρόκληση 5

Το πρόγραμμα καλεί την κλήση συστήματος **fcntl()** με το flag F_GETFD για να διαβάσει τα flags του file descriptor 99, ο οποίος δεν υπάρχει.

Γράφουμε κώδικα που δημιουργεί τον file descriptor αυτόν με την κλήση συστήματος **dup2()**, η οποία μας επιτρέπει να ορίσουμε συγκεκριμένο file descriptor, και τρέχει το πρόγραμμα ως διεργασία-παιδί ωστε να κληρονομήσει το 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!")
```

Εναλλακτικά μπορούμε να εμφυτέψουμε ψευδή τιμή επιστροφής 0 για την κλήση συστήματος fcntl() μέσω της strace:

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

Πρόκληση 6

Αν τρέξουμε το πρόγραμμα με **strace -f** φαίνεται πως δημιουργεί 2 διεργασίες-παιδιά.

Είναι ξεκάθαρο απο τα μυνήματα της πρόκλησης αλλά και τις κλήσεις συστήματος που φαίνονται στην strace οτι οι διεργασίες-παιδιά προσπαθούν να επικοινωνήσουν μεταξύ τους μέσω των file descriptors 33, 34, 53 και 54.

Γράφουμε κώδικα που δημιουργεί **Unix pipes**, τα οποία επιτρέπουν την ενδο-διεργασιακή επικοινωνία. Απο τον κώδικα μας τρέχουμε το πρόγραμμα ως διεργασία-παιδί:

```
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!")
```

Πρόκληση 7

Το πρόγραμμα εκτελεί την κλήση συστήματος **lstat()** για να διαβάσει την κατάσταση των αρχείων .hey_there και .hello_there.
Το αρχείο .hey_there δεν υπάρχει οπότε το δημιουργούμε. Όμως, η πρόκληση πάλι αποτυγχάνει.

Καταλαβαίνουμε από τα υπονοούμενα της πρόκλησης οτι συγκρίνει την κατάσταση των δύο αρχείων και θέλει να είναι όμοια. Για αυτό, δημιουργούμε το .hey_there ως **hard link** του .hello_there:

```
ln .hey_there .hello_there
```

Πρόκληση 8

Το πρόγραμμα θέλει να διαβάσει από ένα απλό αρχείο με όνομα bf00 τον χαρακτήρα X σε ένα συγκεκριμένο σημείο μέσα στο αρχείο, συγκεκριμένα στο byte 1073741824.

Δεν μπορούμε να δημιουργήσουμε ένα τέτοιο αρχείο χειροκίνητα λόγω περιορισμών μνήμης, επομένως γράφουμε κώδικα που θα κάνει τη δουλειά.

Έπειτα από αυτή την προσπάθεια, συνειδητοποιούμε οτι το πρόγραμμα κάνει τον ίδιο έλεγχο για όλα τα αρχεία απο bf00 εώς και bf09, και προσαρμόζουμε τον κώδικα μας:

```
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!")
```

Πρόκληση 9

Το πρόγραμμα προσπαθεί να συνδεθεί σε τοπικό Linux socket στην θύρα 49842. Στην αρχή γράφουμε κώδικα που ακούει και εκτυπώνει εισερχόμενα μηνύματα στο socket αυτό.

Διαπιστώνουμε πως το πρόγραμμα θέλει ο δέκτης του μηνύματος να επικοινωνήσει στον πομπό (στο πρόγραμμα) την απάντηση σε μια τυχαία μαθηματική πράξη.

Προσαρμόζουμε τον κώδικα μας ώστε να λειτουργεί ταυτόχρονα με το πρόγραμμα και να λύνει κάθε μαθηματική έκφραση:

```
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)
```

Προκλήσεις 10 & 11

Το πρόγραμμα δημιουργεί το memory-mapped αρχείο secret_number. Μετά καλεί την κλήση συστήματος **unlink()**, η οποία διαγράφει το όνομα του αρχείου από το file system, αλλά όχι και το ίδιο το αρχείο.
Το αρχείο θα διαγραφεί μόνο όταν κλείσει και ο τελευταίος file descriptor που δείχνει σε αυτό.

Έπειτα το πρόγραμμα φαίνεται να γράφει στο file descriptor του αρχείου την απάντηση στο ερώτημα της πρόκλησης

Επειδή το αρχείο είναι memory-mapped και απο την mmap() έχουμε τη διεύθυνση, μπορούμε να βρούμε τις απαντήσεις στα map files της διεργασίας.
Τρέχουμε το πρόγραμμα σε ξεχωριστό screen:

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

Το map file που μας αφορά συνήθως αναπαρίσταται ως κόκκινο στην εντολή **ls**.

Πρόκληση 12

Το πρόγραμμα δημιουργεί ένα προσωρινό αρχείο στον φάκελο /tmp, του οποίου μέρος του ονόματος είναι τυχαίο.

Μας δίνεται περιορισμένος χρόνος να γράψουμε σε αυτό το αρχείο έναν τυχαίο χαρακτήρα που μας δίνει το πρόγραμμα στο byte 111.

Γράφουμε κώδικα που να βρίσκει το αρχείο αυτό αυτόματα και να γράφει σε αυτό χαρακτήρα που του δίνεται από την γραμμή εντολών ως παράμετρος:

```
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()
```

Πρόκληση 13

Το πρόγραμμα στην αρχή προσπαθεί να ανοίξει το αρχείο .hello_there με δικαιωματα αναγνωσης και εγγραφής. Επειδή απο την 2η πρόσκληση έχουμε αφαιρέσει από τον χρήστη το δικαίωμα εγγραφής στο αρχείο, το επαναφέρουμε:

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

Τρέχουμε ξανά το πρόγραμμα μέσω strace. Αυτή την φορά συνειδητοποιούμε πως δημιουργεί ένα memory map για τον αρχείο, αφου μεταβάλει το μέγεθος του σε 32768 bytes με την χρήση της κλήσης συστήματος **ftruncate()**.

Έπειτα, μειώνει το μέγεθος του και πάλι με την χρήση της ftruncate(). Στη συνέχεια μας ζητάει να γράψουμε στο αρχείο από stdin.

Προφανώς οποιαδήποτε προσπάθεια εγγραφής στο αρχείο θα σηκώσει **SIGBUS**. Πρέπει να μεταβάλουμε το μέγεθος του αρχείου στο αρχικό, και μετά να γράψουμε σε αυτό. Γράφουμε κώδικα:

```
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)
```

Πρόκληση 14

Προφανώς το πρόγραμμα καλεί την **getpid()** με σκοπό να ελέγξει πως έχει το συγκεκριμένο PID 32767. Αν δεν το έχει, η πρόκληση αποτυγχάνει.

Η ανάθεση PID σε διεργασίες απο το λειτουργικό σύστημα γίνεται βάσει του αρχείου **/proc/sys/kernel/ns_last_pid**, το οποίο περιέχει το τελευταίο PID που ανατέθηκε σε διεργασία.

→ Αν αλλάξουμε το περιεχόμενο αυτού του αρχείου πριν τρέξουμε το riddle μπορούμε να ελέγξουμε το PID που θα ανατεθεί στη διεργασία του.

Πρέπει όμως να είμαστε προσεκτικοί: μπορεί μια άλλη διεργασία να δημιουργηθεί μεταξύ της αλλαγής στο αρχείο και την δημιουργία της διεργασίας του riddle.

Το πρόβλημα αυτό λύνεται με την χρήση της κλήσης συστήματος **flock()**, η οποία μας επιτρέπει να κλειδώσουμε ένα αρχείο αποκλειστικά για ανάγνωση/εγγραφή απο την καλούσα διεργασία.

Προτού δούμε των κώδικα αξίζει να σημειώσουμε πως η πρόκληση αυτή πολύ εύκολα λύνεται όπως και η πρόκληση 5, εμφυτεύοντας ψευδή τιμή επιστροφής για την κλήση συστήματος getpid() μέσω strace:

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

```
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")
```