Relazione Homework
Laboratorio di Analisi dei Sistemi

Unico partecipante:
Crosara Marco VR434403

Dicembre 2018 / Gennaio 2019

UNIVERSITÀ DEGLI STUDI DI VERONA
Anno Accademico 2018/2019

Indice

0 Introduzione

A seguito verranno enunciati gli esercizi svolti per l'Homework del Laboratorio di Analisi dei Sistemi. Per ogni esercizio verranno spiegati i file e il codice realizzato per la sua implementazione, i comandi da eseguire per la corretta esecuzione e gli output di programma e dei comandi relativi ai vari esempi.

Tutto il codice Python trattato è stato scritto o convertito nella versione 3.

1 Esercizio 1

Testo dell'esercizio. *Implementare dei test di unità adeguati a verificare la correttezza delle procedure viste a lezione per il calcolo del codice fiscale, in modo tale da ottenere una copertura dei sorgenti quanto più vicina al 100%. Motivare adeguatamente l'eventuale mancato raggiungimento del 100% di copertura.*

1.1 File dell'esercizio

- *codicefiscale.py* : programma visto a lezione per il calcolo del codice fiscale
- *codicefiscale-test.py* : file con gli unit test
- *.coverage* : risultati coverage

1.2 Considerazioni

Il codice di *codicefiscale.py* è quello visto a lezione, tuttavia è leggermente modificato poiché ne è stato fatto il porting del codice da Python 2 a Python 3. Il codice presenta degli errori che sono già stati riscontrati durante la lezione: ad esempio vi sono problemi se il cognome è troppo corto, il nome è troppo lungo o contiene uno spazio, oppure se la persona è nata il giorno 31 di qualsiasi mese.

1.3 Unit Testing

É stato realizzato il file di unit testing *codicefiscale-test.py* che fa il testing del *main* e di tutte le procedure ausiliarie: *estrai_nome_cognome*, *genera_mese*, *codice_comune*, *genera_giorno*, *genera_codice_controllo*.

Gli unit test da implementare sono pochi poiché fin da subito si notano dei risultati discrepanti sull'output delle procedure che permettono di dichiarare il software come non corretto. Gli unit test test implementati che vengono 'falliti' sono 5.

I test presenti sul file consentono comunque di ottenere il 100% della copertura. É stato aggiunto il flag *#pragma: no cover* sul costrutto if relativo alla distinzione del file eseguito come script a se stante o richiamato come modulo, infatti tale ramo di esecuzione non risulta utile ai fini della verifica di copertura.

A seguito i risultati sulla verifica di copertura 100% del codice.

```
mark@mark-XPS:~/.../progetto-finale/1$ python3-coverage report -m
Name                     Stmts   Miss  Cover   Missing
--------------------------------------------------------
codicefiscale-test.py       59      0   100%
codicefiscale.py            72      0   100%
--------------------------------------------------------
TOTAL                      131      0   100%
```

L'esercizio si può considerare svolto con successo poiché si è riusciti tramite i test a dimostrare che il programma *codicefiscale.py* è errato. Prima di svolgere ulteriori test per rilevare eventuali altri errori bisognerebbe correggere le procedure errate (non richiesto dalla consegna) e successivamente effettuare nuovi test su queste ultime. Siamo infine stati in grado di raggiungere la coverage 100%.

2 Esercizio 2

Testo dell'esercizio. *Implementare in Python una procedura non ricorsiva per il calcolo dell'IRPEF. Implementare quindi dei test di unità adeguati a verificare la correttezza della procedura implementata, in modo tale da ottenere una copertura dei sorgenti quanto più vicina al 100%. Motivare adeguatamente l'eventuale mancato raggiungimento del 100% di copertura.*

2.1 File dell'esercizio

- *irpef.py* : programma realizzato per il calcolo dell'irpef
- *irpef-test.py* : file con gli unit test
- *.coverage* : risultati coverage

2.2 Considerazioni

Il programma *irpef.py* riceve in input un argomento passato tramite riga di comando che rappresenta il reddito lordo annuo e ritorna l'aliquota applicata e la tassazione. La procedura implementata sfrutta il tipo di dato Decimal per fare un calcolo più preciso della tassazione da applicare al reddito, il calcolo da effettuare per ottenere la tassa è $reddito * aliquota - correttivo\_fisso$. L'aliquota e il correttivo fisso vengono scelti in base allo scaglione consultando una tabella, rappresentata nel codice come dizionario. L'arrotondamento viene effettuato all'unità di euro per il saldo annuo e al centesimo di euro per la tassa irpef.

Mostriamo ora un esempio di esecuzione del programma *irpef.py*

```
mark@mark-XPS:~/.../2$ python3 irpef.py 15011
Sul tuo reddito di 15011 è stato  applicato il 0.27% e un corretti
vo di 600 euro
La quota irpef risulta quindi di 3452.97 euro
```

2.3 Unit Testing

Il file di unit test *irpef-test.py* testa la procedura realizzata sui diversi scaglioni. Per effetto che nel codice è stato fatto l'uso del dizionario, che rappresenta la tabella con gli scaglioni, anche un solo unit test porta la coverage al 100%. Se fossero stati usati molteplici costrutti if al posto del dizionario questo fatto non si sarebbe verificato. Anche in questo caso, come nell'esercizio precedente, è stato aggiunto il flag *#pragma: no cover* sul costrutto if del file come script o richiamato come modulo.

A seguito vengono mostrati i risultati degli unit tests e della verifica di copertura 100% del codice.

```
mark@mark-XPS:~/.../2$ python3-coverage run irpef-test.py
    [...]
----------------------------------------------------------------------
Ran 8 tests in 0.001s
OK

mark@mark-XPS:~/.../progetto-finale/2$ python3-coverage report -m
Name            Stmts   Miss  Cover   Missing
-----------------------------------------------
irpef-test.py      33      0   100%
irpef.py           18      0   100%
-----------------------------------------------
TOTAL              51      0   100%
```

Su questo esercizio è stata fatta la scelta di scrivere una procedura che consideri in input solo il reddito complessivo annuo, tuttavia online si possono trovare degli strumenti di calcolo che richiedono anche altri dati relativi all'abitazione o agli oneri deducibili. L'esercizio si può considerare svolto con successo poiché siamo riusciti, indipendentemente dalla realizzazione di una procedura per il calcolo dell'irpef più o meno completa, ad implementare un programma corretto e a mostrare tale correttezza tramite l'ausilio degli unit test che hanno raggiunto la copertura del 100% del codice.

3 Esercizio 3

Testo dell'esercizio. *Implementare in Python una procedura per il calcolo della Pasqua. Implementare quindi dei test di unità adeguati a verificare la correttezza della procedura implementata, in modo tale da ottenere una copertura dei sorgenti quanto più vicina al 100%. Motivare adeguatamente l'eventuale mancato raggiungimento del 100% di copertura.*

3.1 File dell'esercizio

– *pasqua.py* : programma realizzato per il calcolo della Pasqua dato l'anno

– *pasqua-test.py* : file con gli unit test

– *pasqua-test-complete.py* : file con altri gli unit test che utilizza un database di riferimento per testare tutte le date del giorno di pasqua per ogni anno

– *DB_pasqua_confronto.txt* : database di riferimento con le date corrette del giorno di pasqua per ogni anno

– *.coverage* : risultati coverage

3.2 Considerazioni

Come da specifiche è stato realizzato un programma Python con una procedura che consente il calcolo del giorno di Pasqua, dato in input come argomento l'anno. Per il calcolo si dovrebbero usare teoricamente due fenomeni: l'equinozio di primavera e le fasi lunari, tuttavia solitamente quando se ne deve implementare l'algoritmo si usa il metodo di Gauss [2] [1]. Quest'ultimo consente di ottenere la data di Pasqua tramite il calcolo di alcuni coefficienti e l'uso di una opportuna tabella, vi sono inoltre delle eccezioni su alcuni anni che vanno gestite opportunamente nel codice. Per ulteriori specifiche implementative si demanda ai commenti sul programma e alla intuitività del codice Python . Mostriamo ora un esempio di esecuzione del programma *pasqua.py*

```
mark@mark-XPS:~/.../progetto-finale/3$ python3 pasqua.py 2018
La Pasqua è il giorno 1 Aprile 2018
```

3.3 Unit Testing

Successivamente alla realizzazione di *pasqua.py*, sono stati realizzati i rispettivi file di unit test *pasqua-test.py* e *pasqua-test-complete.py*. Il primo testa la procedura di calcolo della Pasqua su alcuni casi, finalizzando i test alla coverage 100%, la seconda confronta invece i risultati ottenuti dalla funzione con quelli di un database affidabile trovato online per verificare la correttezza di ogni singolo anno in input accettato dal programma (anni dal 1583 al 2599). A seguito l'esecuzione dei vari unit test con rispettiva verifica della coverage.

Riguardo a *pasqua-test.py*:

```
mark@mark-XPS:~/.../3$ python3-coverage run pasqua-test.py
>> Avvio i test ...
Anno non valido: inserire un anno compreso tra il 1583 e il 2599
.La Pasqua è il giorno 19 Aprile 1609
.La Pasqua è il giorno 27 Marzo 2016
.La Pasqua è il giorno 1 Aprile 2018
.Anno non valido: inserire un anno compreso tra il 1583 e il 2599
.... Test conclusi <<
----------------------------------------------------------------------
Ran 5 tests in 0.000s
OK
```

```
mark@mark-XPS:~/.../progetto-finale/3$ python3-coverage report -m
Name             Stmts   Miss  Cover   Missing
------------------------------------------------
pasqua-test.py      24      0   100%
pasqua.py           26      0   100%
------------------------------------------------
TOTAL               50      0   100%
```

Riguardo a *pasqua-test-complete.py*:

```
mark@mark-XPS:~.../3$ python3-coverage run pasqua-test-complete.py
>> Avvio i test ...
    [...]
Effettuati 1017 test (con anni validi)
.... i test sono stati conclusi <<
----------------------------------------------------------------------
Ran 3 tests in 0.125s
OK

mark@mark-XPS:~/.../progetto-finale/3$ python3-coverage report -m
Name                       Stmts   Miss  Cover   Missing
----------------------------------------------------------
pasqua-test-complete.py       27      0   100%
pasqua.py                     26      0   100%
----------------------------------------------------------
TOTAL                         53      0   100%
```

Si sarebbe potuto migliorare ulteriormente la procedura aumentando il range di anni accettati in input, tuttavia la cosa non risulta essere di grande utilità. Come nel caso dell'esercizio precedente possiamo affermare di aver soddisfatto quanto richiesto dalla consegna poiché si è implementato un programma corretto, con degli unit test più che adeguati a verificarne la correttezza e a farne raggiungere la coverage 100%.

4 Esercizio 4

Testo dell'esercizio. *Utilizzando gli strumenti di analisi dinamica introdotti nel corso del laboratorio, verificare la possibilità di monitorare i fork di processo ed analizzare lo scambio di messaggi tra processo padre e figlio. Verificare la possibilità di furto di dati sensibili, quali credenziali di accesso, ad esempio effettuando tale attività sul processo sshd.*

4.1 File dell'esercizio

- *fork_program.c* : programma di test che esegue il fork di processo
- *fork_program* : compilato del precedente

– *fork_program.py* : corrispettivo Python di *fork_program.c* (serve solo per confronto di codice, non ha alcuna finalità per l'esercizio)

– *sshd.log* : log generato usando `strace` sul processo sshd

4.2 Considerazioni e Realizzazione

É stato realizzato il programma *fork_program.c* che una volta eseguito il fork emula un trasferimento su pipe di informazioni sensibili non criptate tra il processo padre e il processo figlio. Nello specifico se il fork ha successo il figlio richiede all'utente la password per l'avvio di un ipotetico servizio e invia al padre tale password inserita assieme a quella corretta (hardcoded sul codice). Quando le due password arrivano al padre tramite la pipe, quest'ultimo le compara e se sono uguali procede ad avviare il servizio, altrimenti termina. Questo programma, chiaramente vulnerabile per diversi motivi, ci serve solo a capire se sia possibile monitorare i fork e la comunicazione padre/figlio.

Procediamo all'analisi di *fork_program* tramite il comando `strace` con l'opzione `-e` in particolare andiamo a tracciare 'process', 'read' e 'write'. Aggiungiamo infine l'opzione `-f` che come suggerito dal manuale ci consente di seguire i fork di processo. L'output del comando sarà il seguente (vengono mostrate solo le parti a noi utili).

```
mark@ma.../4$ strace -f -e trace=process,read,write ./fork_program
    [...]
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SET
TID|SIGCHLD, child_tidptr=0x7f41315bf7d0) = 20341
    strace: Process 20341 attached
[pid 20340] read(3,  <unfinished ...>
[pid 20341] write(1, "FIGLIO: Inserisci la password pe"..., 55FIGL
IO: Inserisci la password per avviare il servizio: ) = 55
[pid 20341] read(0, jerry11@"jerry11@\n", 1024) = 9
[pid 20341] write(15, "jerry11@|mario1234\0", 19 <unfinished ...>
[pid 20340] <... read resumed> "jerry11@|mario1234\0", 111) = 19
    [...]
```

Come possiamo notare è possibile chiaramente distinguere il momento in cui il processo padre ha richiesto il fork tramite la system call 'clone', vediamo inoltre il nuovo pid [pid 20341] che viene ritornato dal sistema e assegnato al processo figlio. Il nuovo processo chiede all'utente di inserire la password per l'avvio del servizio e quando l'ha ottenuta la invia sulla pipe al padre assieme a quella corretta (`write(15, "jerry11@|mario1234", 19 <unfinished ...>`), nella riga esattamente successiva abbiamo invece la lettura dalla pipe da parte del padre. In questo momento, monitorando i processi, riusciamo dunque a carpire la password corretta.

4.3 sshd

Passiamo ora all'analisi del processo *sshd* e alla eventuale possibilità di furto di dati sensibili. Se non fosse installato Secure Shell (SSH) procediamo all'installazione tramite il comando `sudo apt install openssh-server`, una volta fatto questo verifichiamo che tutti i servizi siano correttamente avviati tramite il comando seguente:

```
mark@mark-XPS:~/.../4$  sudo service ssh status
  ssh.service - OpenBSD Secure Shell server
  Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor
  preset: enabled)
    [...]
 Main PID: 1096 (sshd)
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/ssh.service
           1096 /usr/sbin/sshd -D
    [...]
```

Da questo comando leggiamo il pid del processo sshd: 1096 e lo usiamo per avviare il monitoraggio con `sudo strace -f -p 1096 -o sshd.log`. Rispetto a prima abbiamo aggiunto l'opzione `-p` per monitorare tramite pid un determinato processo già in esecuzione e l'opzione `-o` per chiedere a strace di fare il log sul file *sshd.log*. Procediamo infine con la simulazione di una eventuale connessione ssh, eseguendo su un altro terminale il comando `ssh username@localhost`, ci verrà richiesta la password dell'utente e alla conferma della stessa verrà dunque avviata la shell remota. Terminiamo ora il monitoraggio di sshd e analizziamo i log generati.

Il file *sshd.log* è stato lasciato per intero per la consultazione (con una ovvia modifica alla password), vedremo ora solamente le parti che ci interessano.

```
    [...]
1096  clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7fa2f2f3abd0) = 21578
    [...]
21578 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f87dff78bd0) = 21579
    [...]
21579 write(4, "\0\0\0\rPerryThePlatypus", 17 <unfinished ...>
    [...]
21578 read(6, "\f\0\0\0\rPerryThePlatypus", 18) = 18
    [...]
```

Il processo padre sshd con id 1096 effettua molte operazioni ed esegue molti fork, tuttavia ci interessano in particolare quelli riportati qui sopra: possiamo infatti notare che viene avviato il processo figlio 21578 che a sua volta effettuerà una clonazione e genererà il processo figlio 21579. Senza approfondire ulteriormente la cosa e osservando gli ultimi due log riportati qui sopra possiamo

intuire che 21578 e 21579 si occupino tra le altre cose dell'autenticazione alla shell remota e possiamo in particolare notare che si scambiano la password in chiaro.

Concludiamo quindi che si può dare risposta positiva alla domanda sull'esistenza del rischio di furto di dati sensibili, quali credenziali di accesso, tramite il monitoraggio del processo sshd. Abbiamo infatti dimostrato che tramite l'utilizzo del comando `strace` è possibile carpire la password di login alla shell remota.

5 Esercizio 5a

Testo dell'esercizio. *Implementare una procedura che tenta di scrivere in un file posto in una directory per cui non è concessa autorizzazione di scrittura o accesso (e.g. `/var`). Utilizzando gli strumenti di analisi dinamica introdotti nel corso del laboratorio, verificare la possibilità di monitorare tali tentativi di scrittura e gestirli generando un log.*

5.1 File dell'esercizio

- *sum_malware.c* : programma che tenta di scrivere su un file esistente in `/var`
- *sum_malware* : compilato del precedente
- *hello_name.c* : programma che tenta di creare un nuovo file in `/var`
- *hello_name* : compilato del precedente
- *eacces.log* : file di log che registra tutti i tentativi falliti di accesso a un file per mancanza di autorizzazione di lettura/scrittura
- *AnalyzerPathAccess.java* : programma scritto in java che consente di monitorare gli accessi a file che si trovano in directory per cui non si dispone dei diritti. *AnalyzerPathAccess.java* genera inoltre il file di log
- *AnalyzerPathAccess.class*

5.2 Considerazioni e Realizzazione

Per prima cosa è stato realizzato *sum_malware.c*, esso rappresenta un programma che oltre al funzionamento consueto di svolgere la somma tra due numeri interi dati in input come argomenti, presenta anche un comportamento malevolo. Nello specifico prima di terminare esso chiama una procedura che tenta di scrivere in un file di una directory per cui non è concessa autorizzazione di scrittura o accesso (nel nostro caso `/var`). Tentando di eseguire lo stesso, dopo aver ottenuto il risultato della somma, avremo anche un 'Segmentation fault' che corrisponde al tentativo fallito di accedere in scrittura al file `/var/run/acpid.pid`.

Si noti che se si esegue *sum_malware* su un elaboratore che non ha il file sopra citato nella cartella `/var/run/`, il risultato dell'esecuzione potrebbe essere diverso e potrebbe non verificarsi l'errore di accesso negato. Qui sotto l'esempio.

```
mark@mark-XPS:~/.../progetto-finale/5a$ ./sum_malware 8 22
SOMMA DI DUE NUMERI
La somma dei numeri in input è: 30
Segmentation fault (core dumped)
```

L'idea è che per poter monitorare questi accessi basti utilizzare il comando `strace`, restringendo l'output con l'opzione `-e trace=openat` e in particolare ricercando tra i vari log generati dal comando il codice di errore EACCES. Tale errore identifica appunto una negazione del permesso di accesso ad un file per cui non si dispone delle dovute autorizzazioni.

```
mark@mark-XPS:~/.../5a$ strace -e trace=openat ./sum_malware 8 22
    [...]
SOMMA DI DUE NUMERI
La somma dei numeri in input è: 30
openat(AT_FDCWD, "/var/run/acpid.pid", O_WRONLY|O_CREAT|O_TRUNC,
0666) =-1 EACCES (Permission denied)
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=NULL}
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
```

Il passo finale è stato la realizzazione di un programma java che tramite l'esecuzione del comando sopra citato si occupi di popolare un log *eaccess.log* con data, ora e l'indicazione del tentativo fallito di accesso a un file per cui non si dispone dei diritti. L'idea è di richiedere tramite il programma *AnalyzerPathAccess.java* una esecuzione del comando `strace -ttfe trace=openat ./sum_malware 1 2` e di ricercare tra i vari log così ottenuti una negazione sulla richiesta di apertura di un file per cui non si dispone dei diritti: tale negazione corrisponde come già detto all'errore 'EACCES'. Come si può vedere il comando utilizzato dall'analizzatore ha delle opzioni in più su strace rispetto all'esempio fatto prima: aggiungiamo infatti l'opzione `-f` per seguire eventuali fork e `-tt` per aggiungere il timestamp ad ogni log. Ulteriori specifiche in merito al funzionamento di *AnalyzerPathAccess.java* si demandano ai commenti sul codice, qui sotto un esempio di utilizzo.

```
mark@mark-XPS:~/.../5a$ java AnalyzerPathAccess ./sum_malware 1 2
eacces.log aggiornato con 1 nuovo messaggio di log
```

Una volta eseguito il programma verrà aggiunta una nuova riga al file di log con il nuovo tentativo fallito. Qui a seguito un esempio di una riga di log generata sul file *eacces.log*.

```
2019-01-08 15:05:57 ./sum_malware /var/run/acpid.pid 15:05:57.3641
25 openat(AT_FDCWD, "/var/run/acpid.pid", O_WRONLY|O_CREAT|O_TRUNC
, 0666) = -1 EACCES (Permission denied)
```

Analizziamo il messaggio di log:

- `2019-01-08 15:05:57` : data e ora in cui *AnalyzerPathAccess.java* ha salvato il log sul file
- `./sum_malware` : programma analizzato da *AnalyzerPathAccess.java*
- `/var/run/acpid.pid` : file a cui è stato negato l'accesso poiché non si dispone dei diritti
- `15:05:57.364125` : ora:minuti:secondi.millesimi in cui strace ha generato il log (disponibile grazie all'opzione `-tt` su strace). Si noti che per programmi con una esecuzione molto lunga questo tempo potrebbe differire dal primo, infatti prima strace genera il log e solo successivamente *AnalyzerPathAccess.java* lo salva su *eacces.log*
- `openat(AT_FDCWD, "/var/run/acpid.pid", O_WRONLY|O_CREAT|O_TRUNC, 0666) = -1 EACCES (Permission denied)` : messaggio di log strace originale

Oltre a *sum_malware.c* è stato scritto anche il programma *hello_name.c*, quest'ultimo è simile al primo ma differisce dal fatto che dopo aver preso in input come argomento un nome e dopo aver salutato l'utente, esegue il comportamento 'malevolo' di tentare la creazione di un file *nome.txt* su `\var`. Possiamo dunque dare in pasto ad *AnalyzerPathAccess.java* anche questo programma, il log ottenuto sarà simile a quello precedente. Qui sotto il l'esecuzione e il messaggio di log risultante.

```
mark@mark-XPS:~/.../5a$ java AnalyzerPathAccess ./hello_name marco
eacces.log aggiornato con 1 nuovo messaggio di log
```

```
2019-01-08 17:43:59 ./hello_name /var/marco.txt 17:43:59.236726
openat(AT_FDCWD, "/var/marco.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666)
= -1 EACCES (Permission denied)
```

L'esercizio si può considerare svolto con successo poiché tramite l'uso del comando `strace` visto a lezione, è stato possibile realizzare un programma che consente di monitorare e fare il log dei tentativi di accesso a un file posto in una cartella di cui non si dispone dei diritti di lettura/scrittura.

6 Esercizio 5b

Testo dell'esercizio. *Implementare in C una procedura che accede ad aree di memoria scelte in modo casuale. Utilizzando gli strumenti di analisi dinamica introdotti nel corso del laboratorio, verificare la possibilità di monitorare i tentativi di accesso ad aree di memoria non allocate al processo e gestirli generando un log.*

6.1 File dell'esercizio

- *random_memory.c* : programma molto semplice che tenta di accedere a un indirizzo di memoria casuale per cui non dispone dei diritti
- *random_memory* : compilato del precedente
- *memorydenied.log* : file di log che registra tutti i tentativi falliti di accesso a una zona di memoria.
- *AnalyzerMemoryAccess.java* : programma scritto in java che consente di monitorare gli accessi a zone di memoria per cui non si dispone dei diritti. *AnalyzerMemoryAccess.java* genera inoltre il file di log
- *AnalyzerMemoryAccess.class*

6.2 Considerazioni e Realizzazione

Prima di tutto, seguendo le istruzioni, è stato realizzato il programma *random_memory.c* che ogni volta che viene avviato accede anche ad aree di memoria scelte in modo casuale, questa operazione viene fatta tramite l'utilizzo della procedura *rand()*. La richiesta di accesso viene ovviamente negata dal sistema operativo e provoca dunque la terminazione dell'applicazione con il segnale 'SIGSEGV':

```
mark@mark-XPS:~/.../progetto-finale/5b$ ./random_memory
PROGRAM ADDRESS: 0x55b8e60db260
              >>: 5
RANDOM ADDRESS: 0x25afb298
Segmentation fault (core dumped)
```

Il prossimo passo è rilevare l'accesso alle aree di memoria non allocate al processo, dunque le zone di memoria per cui *random_memory* non dispone dei permessi. L'idea per procedere è simile a quella del precedente esercizio: si utilizza `strace` per rilevare il Segmentation fault e per osservare se quest'ultimo avviene a causa dell'accesso negato a un indirizzo di memoria, se questo fatto è verificato allora salviamo i risultati acquisiti in un messaggio su un file di log. Partiamo quindi dal precedente programma di analisi realizzato e lo modifichiamo per adattarlo a questo nuovo esercizio. Con poche modifiche al codice si ottiene il programma *AnalyzerMemoryAccess.java* che a differenza del precedente osserva se strace ritorna un log contenente un 'SIGSEGV'. Avendo tali messaggi basta ora verificare che il Segmentation fault sia causato da l'accesso a un indirizzo di memoria. Per fare questa verifica, nella procedura *logIntegration(...)* controlliamo che il campo `si_addr` sia popolato con un indirizzo, in tal caso procediamo alla composizione del messaggio di log che *AnalyzerMemoryAccess* aggiungerà in append al file *memorydenied.log*. Per ulteriori specifiche implementative si demanda al solito ai commenti sul codice.

Qui sotto l'esecuzione di *AnalyzerMemoryAccess* su *random_memory* e il relativo messaggio di log generato e salvato sul file.

```
mark@mark-XPS:~/.../5a$ java AnalyzerMemoryAccess ./random_memory
memorydenied.log aggiornato con 1 nuovo messaggio di log

2019-01-08 22:31:41 ./random_memory 0x52944b80 22:31:40.851871 ---
SIGSEGV{si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x52944b80}
```

Analizziamo il messaggio di log:

- `2019-01-08 22:31:41` : data e ora in cui l'analizzatore java ha salvato il messaggio di log sul file
- `./random_memory` : programma analizzato
- `0x52944b80` : indirizzo di memoria a cui il sistema operativo ha negato l'accesso e che ha causato il Segmentation fault
- `22:31:40.851871` : ora:minuti:secondi.millesimi in cui strace ha generato il log (`-tt`). Come nell'esercizio precedente, si noti che su programmi con una esecuzione molto lunga questo tempo potrebbe differire dal primo.
- `SIGSEGV (si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x52944b80)` : messaggio di log strace originale

L'esercizio si può considerare svolto con successo poiché tramite `strace` è stato possibile realizzare un analizzatore java che consente di fare il log dei tentativi di accesso a indirizzi di memoria non allocate al processo.

7 Considerazioni conclusive

In alcuni esercizi si sarebbe potuto discutere l'utilizzo di `ltrace` anziché `strace` tuttavia abbiamo preferito il primo per comodità e continuità di utilizzo.

A riguardo degli esercizi 5a e 5b: *AnalyzerPathAccess.java* e *AnalyzerMemoryAccess.java* sono programmi pressoché uguali, infatti entrambi svolgono la funzione di ricerca di particolari log offerti da strace. Si sarebbe quindi potuto realizzare un unico programma di analisi per gli esercizi 5a e 5b oppure realizzare una libreria con le funzioni comuni, questo non è stato fatto per una logica di separazione degli esercizi.

Riferimenti bibliografici

[1] http://calendario.eugeniosongia.com/gauss.htm . metodo aritmetico di gauss per il calcolo della data della pasqua.

[2] http://e-ware.org/blog/algoritmo-calcolo-della-pasqua/ . Algoritmo calcolo della pasqua.

[3] https://linux.die.net/man/1/strace . strace(1) - linux man page [corrispet tivo di 'man strace'].

[4] https://linux.die.net/man/8/sshd . sshd(8) - linux man page [corrispettivo di 'man sshd'].