Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Analisi di Sistemi Informatici

Relazione degli Homework di Laboratorio

Candidati:
Riccardo Astolfi
Giacomo Ferro
Francesco Gobbi

Anno Accademico 2018/2019

Indice

Introduzione **2**

Esercizio 1 - Calcolo del codice fiscale **3**
Codice dell'algoritmo 3
Testing 4

Esercizio 2 - Calcolo dell'Irpef **5**
Codice dell'algoritmo 5
Testing 5

Esercizio 3 - Calcolo della Pasqua **6**
Codice dell'algoritmo 6
Testing 6

Esercizio 4 - Monitoraggio dei processi **7**

Esercizio 5a (semplificato) - Monitoraggio degli accessi in memoria non autorizzati **9**

Esercizio 5b (complesso) - Monitoraggio degli accessi in memoria randomica, tramite puntatore **10**

Introduzione

Nella seguente relazione sono riportati gli esercizi svolti dal nostro gruppo e le relative analisi effettuate su di essi.

Abbiamo scelto di svolgere tutti gli esercizi assegnati.

- **Esercizio 1**: implementazione dei test di unità adeguati a verificare la correttezza delle procedure 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.

- **Esercizio 2**: implementazione in Python di una procedura non ricorsiva per il calcolo dell'IRPEF e, in seguito, 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.

- **Esercizio 3**: implementazione in Python di una procedura per il calcolo della Pasqua e, in seguito, 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.

- **Esercizio 4**: Utilizzando gli strumenti di analisi dinamica introdotti nel corso del laboratorio, sono stati monitorati i fork di processo ed analizzato lo scambio di messaggi tra processi padre e figlio. È stata monitorata la possibilità di furto di dati sensibili, quali credenziali di accesso, ad esempio effettuando tale attività sul processo *sshd*.

- **Esercizio 5a**: implementazione di 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, sono stati monitorati tali tentativi di scrittura e sono stati gestiti generando un log.

- **Esercizio 5b** 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.

Esercizio 1 - Calcolo del codice fiscale

Codice dell'algoritmo

Per lo svolgimento di questo esercizio ci siamo basati sul codice proposto durante l'ultima lezione per il calcolo del codice fiscale, apportando delle modifiche su casi non inizialmente considerati.

- Abbiamo considerato casi in cui il nome non abbia 4 consonanti.
- Abbiamo considerato i casi in cui il nome ed il cognome non abbiano un numero minimo di 3 consonanti, aggiungendo prima le eventuali vocali e poi x.
- Abbiamo considerato il caso in cui il nome o il cognome siano composti da più di una parola o che al loro interno presentino l'apostrofo ('), andando a considerare consonanti e vocali anche di questi.

Il file *codiceFiscale.py durante l'esecuzione richiede: nome, cognome, data di nascita (gg/mm/aaaa), comune di nascita e sesso (m oppure f).* Questi parametri verranno passati alle seguenti procedure:

- *estrai_ nome(nome)*: applica l'algoritmo per la selezione dei tre caratteri (consonanti e/o vocali) identificativi del nome (con eventuali casi);
- *estrai_ cognome(cognome)*: applica l'algoritmo per la selezione dei tre caratteri (consonanti e/o vocali) identificativi del cognome (con eventuali casi);
- *genera_ mese(mese)*: restituisce il codice corrispondente al mese, preso da una tabella;
- *codice_ comune(comune)*: restituisce il codice corrispondente al comune di nascita, preso da una tabella;
- *genera_ giorno(giorno, sesso)*: restituisce il giorno di nascita se l'individuo è di sesso maschile, altrimenti restituisce il giorno sommato alla costante 40;

- *genera_ codice_ controllo(codice)*: riceve in input un codice fiscale senza l'ultimo carattere e restituisce lo stesso concatenandogli un carattere preso da una tabella, dopo aver effettuato delle operazioni definite dall'algoritmo.

N.B. Abbiamo usato le informazioni di Wikipedia per la generare il Codice Fiscale.

Testing

In seguito abbiamo creato il file *test.py* dove sono presenti gli *unit test* relativi alle procedure descritte.
Per eseguire il test abbiamo usato una suite in cui sono aggiunte tutte le funzioni utilizzate nel file *codiceFiscale.py*. Per ognuna delle funzioni utilizzate si fanno uno o più test usando la tipologia *assertEqual*. Quindi il test sa già quale sarà il risultato vero e lo va a confrontare con il risultato dato dalle funzioni utilizzate dal programma.
Questi test servono per verificare se l'output di ogni singola funzione nel file creato, per un determianto scopo, sono uguali ai risultati aspettati. La selezione degli input non è casuale, in quanto si usano esempi per avere un più alto grado di *coverage* del codice quanto più vicina al 100%. Questo viene fatto trovando dei test in modo tale da eseguire tutti i possibili *branch* nell'esecuzione.

Gli input da noi scelti permettono di effettuare una *coverage* il più possibili vicina al 100% delle procedure di interesse.
Non essendo invece rilevanti i test sul *main* (verrà sempre eseguito all'avvio del programma), con la clausola *"♯pragma: no cover"* questi sono esclusi dal programma per fare la coverege.

Tutti questi test sono possibili tramite l'esecuzione del file *script.sh*, che li eseguirà tutti e riporterà i risultati nel file *report.txt* mostrando le seguenti informazioni relative a *codiceFiscale.py*:

- il numero di statement presenti nel programma (*Stmts*) e quanti non sono stati eseguiti (*Miss*);
- il numero di branch presenti nel programma (*Branch*) e quanti non sono stati visitati (*BrMiss*);
- percentuale di *coverage* dell'intero file (*Cover*);
- le righe del codice non visitate durante l'esecuzione (*Missing*).

Oltre al file .txt, viene creato anche un file di coverage visibile da browser, in formato html, che permette una visualizzazione più semplificata ed intuitiva dei vari branch visitati o meno, delle righe di codice eseguite o meno e molto altro. (**Questa tipologia di test e questa tipologia di visualizzazione è stata utilizzata per tutti i primi 3 esercizi.**)

Esercizio 2 - Calcolo dell'Irpef

Codice dell'algoritmo

Per lo svolgimento di questo esercizio ci siamo basati sull'algoritmo di calcolo dell'*Irpef* prendendo le informazioni da questa pagina web: `https://www.fiscoetasse.com/approfondimenti/12069-scaglioni-e-aliquote-irpef.html`.

Il file *irpef.py* eseguito deve avere come paramentri in input il reddito annuale per calcolare l'*Irpef*.
Il valore inserito deve essere numerico, così non causerà errori nella procedura.
In questo file è usata una sola funzione, ovvero:

- *irpef_ calculation(income)*: calcola l'importo dell'*Irpef* da pagare in base al reddito annuo. Usando 3 vettori in cui sono salvate le fasce di reddito, con le corrispondenti aliquote in percentuale di pagamento per le imposte intemedie e il prezzo da pagare le per imposte precedenti. Tutti questi dati sono presi dal sito web indicato sopra.
  Quindi con un ciclo *for* si va ad iterare le possibili fasce di reddito e a seconda dei casi si entrarà in uno dei 3 *if* presenti. Il calcolo viene fatto in una sola esecuzione, grazie ai dati presenti dai vettori inizializzati da noi.
  Infine il programma ritorna il valore dell'imposta sul reddito inserita in input.

Testing

Per il test è presente il file *test.py* nel quale abbiamo eseguito gli *unit test* relativi a questo programma.
In questo caso siamo andati ad usare gli *assertEqual* per verificare se dati determinati input di reddito, la funzione generava gli unput uguali al risulato vero.
Per risultato vero intendiamo il valore ottenuto di irpef con i medesimi input su questa pagina web: `https://www.avvocatoandreani.it/servizi/calcolo-irpef.php`.

Anche in questo caso la selezione degli input non è stata casuale,ma sempre per ottenere una *coverage* del codice quanto più vicina al 100%. Quindi inserendo dei casi per ogni branch presente nel codice.

Il risultato ottenuto è molto buono, in quanto siamo riusciti ad ottenere il 96% di coverage.

Anche in questo caso abbiamo usato la clausula *"♯pragma: no cover"* per escludere le parte di programma, come nel main(), dal calcolo della copertura.

Utilizziamo come visto precedentemente un file *Script.sh* che eseguirà tutti i test possibili e crearà in automatico i report in vesione *.txt* e in *.html*, come nell'esecizio precedente.

Esercizio 3 - Calcolo della Pasqua

Codice dell'algoritmo

Per lo svolgimento di questo esercizio ci siamo basati sul **metodo di Gauss** per il calcolo della *Pasqua*, presente a questo indirizzo: `https://it.wikibooks.org/wiki/Implementazioni_di_algoritmi/Calcolo_della_Pasqua`. Noi abbiamo preso, ovviamente la versione in Python e semplicemente abbiamo utilizzato questo algoritmo, commentandolo e verificandolo.

Il file *CalcoloPasqua.py* viene eseguito inserendo l'anno di cui si vuole sapere la data della Pasqua. Questo parametro dovrà essere un numero maggiore di 1582, anno in cui è stato introdotto il nostro *calendario gregoriano* e questo sarà poi passato insieme all'algoritmo.

Testing

In seguito abbiamo creato il file *test.py* dove abbiamo eseguito gli *unit test* relativi all'algoritmo. Abbiamo usato ancora gli *assertEqual* per verificare la procedura, con eventuali input.

Ogni input utilizzato per il test è stato confrontato con il valore vero, calcolato in internet, semplicemente ricercando su Google: "Anno Pasqua x", dove al posto della "x" è inserito l'anno desiderato.

Anche qua gli input sono eseguiti per ottene il più alto valore di *coverage*, andando a creare dei casi che consentano di ricoprire il più alto numero di *branch* durante l'esecuzione.

Sul *main* abbiamo aggiunto la clausola *♯pragma: no cover* così escludiamo quella parte di codice dal coverage.

Il file *Script.sh* esegue tutti i test presenti enl file e riporta i vari risultati della coverage nel file *report.txt* e nel file *.htlm*.

Esercizio 4 - Monitoraggio dei processi

In questo esercizio abbiamo creato tre file: *fork.c*, *main.c* e *sysAnalysis.sh*.

- Nel primo file (*fork.c*) c'è la creazione del figlio, il padre scrive nella pipe e il figlio legge.
- Il file *main.c* esegue una fork e poi il processo figlio esegue una exec del file sysAnalysis.sh
- Il file *sysAnalysis.s* viene lanciato dal file *main.c*

Diciamo subito che la *strace* è stata usata con delle opzioni che andranno a modificare il suo output secondo alcuni parametri. Nel nostro codice sono presenti:

- *"-o filename"*: riporta l'output del comando nel file indicato;
- *"-ff"* : se è presente l'opzione *" -o output_filename"*, ogni processo tracciato viene scritto nel file *filename.pid* (dove **pid è il codice identificativo del processo tracciato**)

Procedura:

- All'inizio viene eseguito il file *main.c*, il quale lancia il file *sysAnalysis.sh* che a sua volta compila ed esegue il file *fork.c*.
  La *strace* salva i tracciati nei file *dump.txt.pid*.
- Dopo aver salvato i risultati in un file specifico per ogni pid, si salvano i pid dei processi creati su cui *strace* ha tracciato le chiamate a sistema.
- Si procede a cercare con un ciclo *for()* il pid del padre.
- Successivamente si procede ad analizzare le fork eseguite dal padre (ricercando la chiamata *clone* nel file *dump.txt.pid_padre*).
  Estraiamo i pid dei figli e salviamo le informazioni (pid e numero di figli creati) sul file *fork.log*.
- Infine analizzare su che descrittore ha scritto il padre, attraverso l'opportuna ricerca sul file *dump.txt.pid_padre*, tali informazioni sono scritte sul file *writes.log*
  Il descrittore di riferimento è il numero 4 (in quanto 0,1,2 sono rispettivamente lo *standard input*, *standard output*, *standard error*).
  N.B. I descrittori devono essere strettamente maggiori di due.

Alla fine quindi, avremo salvato su file le informazioni su quanti figli sono stati creati e quali sono i relativi pid, e poi anche i dati relativi alle scritture su pipe.

Sono inoltre presenti due script bash per la prova di furto di dati sul processo *ssh*:

- Il file *open-ssh.sh* avvia una sessione ssh collegandosi a localhost (si poteva indicare anche un indirizzo IP non locale).
  La sessione viene poi chiusa con *exit* per consentire il completamento del processo di intercettazione delle chiamate di sistema.

- Il file *sniffer-ssh.sh* recupera il pid del processo *ssh* da monitorare ed esegue il comando *strace* su di esso generando il file *ssh.log*. Tale script deve essere lanciato prima di inserire la password per localhost nel lancio del primo file.

Esercizio 5a (semplificato) - Monitoraggio degli accessi in memoria non autorizzati

Il seguente esercizio si basa principalmente sull'utilizzo del comando bash *ltrace*.
Esso traccia e permette di salvare le chiamate dinamiche alle librerie, effettuate dal processo in esecuzione e i segnali ricevuti da esso. Questo comando mostra i parametri delle funzioni invocate e le chiamate di sistema.
Abbiamo usato *ltrace* con le seguenti opzioni:

- *-b*: disabilita la stampa dei segnali ricevuti dal processo tracciato;
- *-o "output_filename"*: riporta l'output del comando nel file *output_filename*.

Per lo svolgimento di questo esercizio è stata implementata una procedura che tenta di scrivere in un file posto in una directory per cui non è concessa autorizzazione di scrittura o accesso.
La cartella di riferimento è */var*.
Nel file C *accesso_var.c* si tenta di creare un file *prova.txt* nella directory */var* protetta.
Per tale ragione, a funzione di libreria *fopen("/var/prova.txt", "w")* ritornerà il valore **NULL**.
Il file C *main.c* fa una fork ed esegue attraverso una *exec* lo script bash *checkVar.sh*.
Il file *checkVar.sh* esegue *ltrace* su accessoVar e salva l'output su file. Successivamente si cerca il riferimento al tentativo di apertura di file salvando poi il percorso della cartella.
Dato tale percorso, se il proprietario della cartella *(root)* coincide con il proprietario che ha eseguito il codice, allora si salva l'accesso come accesso consentito, viceversa lo si etichetta come non consentito.
Tale risconto verrà scritto su file *accessi_tracciati.log*.

Esercizio 5b (complesso) - Monitoraggio degli accessi in memoria randomica, tramite puntatore

La procedura è simile a quella dell'esercizio precedente.
Nel file *accessoRandom.c* si istanzia un puntatore con indirizzo di memoria randomizzato attraverso la funzione *rand()*. Lanciando il main si esegue lo script *checkPtr.sh* che esegue strace su *accessoRandom*.
Dal file di traccia si ricerca la chiamata SIGSEGV recuperando l'indirizzo di dove è avvenuto l'errore di segmentazione previsto per aver tentato di accedere ad un'area di memoria non preventivamente allocata.
Tale indirizzo recuperato viene salvato su file *accessi_tracciati.log* come accesso non consentito.