Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Analisi dei Sistemi Informatici

Relazione delle attività di laboratorio

Candidati:

Davide Bianchi
Matricola VR424505

Marco Colognese
Matricola VR423791

Mattia Rossini
Matricola VR423614

Anno Accademico 2017/2018

Indice

Introduzione

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

Essendo un gruppo di 3 membri, abbiamo scelto di svolgere 5 dei 6 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%.

- **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%.

- **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%.

- **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 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. Ad esso abbiamo apportato le seguenti modifiche (facendo riferimento a questa procedura di calcolo: `http://www.calcolocodicefiscale.org/algoritmo+codice+fiscale.html`) per migliorare l'algoritmo che non ricopriva alcuni casi particolari quali:

- implementazione differente degli algoritmi per il calcolo del nome e del cognome (inizialmente calcolati allo stesso modo).

  La differenza sta nella scelta delle consonanti per nomi e cognomi aventi più di quattro lettere:

  - nel caso del nome verranno considerate la prima, la terza e la quarta lettera;
  - nel caso del cognome verranno scelte le prime tre consonanti.

- Calcolo del codice fiscale per individui aventi nome e/o cognome composti da meno di 3 lettere, aggiungendo un numero di volte il carattere x pari alle lettere mancanti.

- Calcolo del codice fiscale per individui aventi nome e/o cognome composti da 3 lettere, due della quali sono la stessa vocale (es. *Ada* viene calcolata come *daa*).

Il file *codiceFiscale.py* deve essere eseguito ricevendo in input cinque parametri: 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;

- *estrai_cognome(cognome)*: applica l'algoritmo per la selezione dei tre caratteri (consonanti e/o vocali) identificativi del cognome;

- *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.

Testing

In seguito abbiamo creato il file *test.py* nel quale sono contenuti gli *unit test* relativi alle procedure sopradescritte. Le parti principali di questi test sono gli *assert*, che si dividono in quattro tipologie:

- *assertEqual*: verifica che la procedura, con eventuali input, restituisca il risultato atteso (nei nostri test abbiamo optato per questa soluzione);

- *assertTrue* o assertFalse: per verificare una condizione booleana;

- *assertRaises*: verifica che venga sollevata una certa eccezione.

Essi sono utili per verificare il corretto funzionamento dell'algoritmo implementato poiché vanno a testare l'output di ciascuna funzione sui vari input scelti. La selezione degli input non è casuale, bensì devono essere delle combinazioni che permettano di effettuare una *coverage* del codice quanto più vicina al 100%. Per fare ciò è necessario inserire dei parametri che consentano di ricoprire ciascun *branch* di esecuzione.

Gli input da noi scelti permettono di effettuare una *coverage* del 100% delle procedure di interesse. Non essendo invece rilevanti i test sul *main* (verrà sempre eseguito all'avvio del programma), abbiamo inserito la clausola ♯*pragma: no cover* che ci consente di escludere la porzione di codice scelta dall'analisi del programma.

Per verificare la correttezza dei test presenti nel file *test.py* è sufficiente avviare il Bash-Script chiamato *script.sh* che li eseguirà tutti e riporterà i risultati nel file *report.txt* mostrando le seguenti informazioni relative a *codiceFiscale.py*:

- numero di statement presenti nel programma (*Stmts*) e quanti di essi non sono stati eseguiti (*Miss*);

- numero di branch presenti nel programma (*Branch*) e quanti di essi non sono stati visitati (*BrMiss*);

- la percentuale di *coverage* dell'intero file (*Cover*);
- le righe di codice non visitate dai test (*Missing*).

Lo script riporterà inoltre il risultato dei test in formato *HTML*. In esso sarà riportato il codice sorgente del programma del quale si potranno evidenziare quattro tipi di informazione attraverso i bottoni presenti nella parte superiore:

- statement eseguiti (*run*) in verde;
- statement non eseguiti (*missing*) in rosso;
- statement esclusi dall'analisi (*excluded*) in grigio;
- statement parzialmente eseguiti (*partial*) in giallo (ad esempio costrutti *if* privi del ramo *else*).

Esercizio 2 - Calcolo dell'Irpef

Codice dell'algoritmo

Per lo svolgimento di questo esercizio ci siamo basati sull'algoritmo di calcolo dell'*Irpef* (acronimo di *Imposta sul Reddito delle Persone Fisiche*) presente a questo indirizzo: `https://www.studiocataldi.it/risorse/calcolo-irpef/`.

Il file *irpef.py* deve essere eseguito ricevendo in input un solo parametro: il reddito annuale sul quale si vuole calcolare l'*Irpef*. Questo parametro, dopo aver constatato che si tratta di un valore numerico positivo, verrà passato alla seguente procedura:

- *irpef_ calculation(income)*: questa funzione si occuperà di analizzare il reddito inserito dall'utente per capire in quale dei cinque scaglioni ci si trova ed applicare l'equazione corretta per il calcolo dell'*Irpef*.

  La procedura infine ritornerà il valore dell'imposta sul reddito relativa all'importo inserito e allo scaglione in cui si troverà l'individuo.

Testing

In seguito abbiamo creato il file *test.py* nel quale sono contenuti gli *unit test* relativi alla procedura sopradescritta. Le parti principali di questi test sono gli *assert* e, anche in questo caso, ci siamo orientati sull'utilizzo degli *assertEqual* per verifica che la procedura, con eventuali input, restituisca il risultato atteso.

Questi test sono stati utili per verificare il corretto funzionamento dell'algoritmo implementato, infatti all'indirizzo `https://www.studiocataldi.it/risorse/calcolo-irpef/` è anche possibile effettuare online il calcolo dell'*Irpef* su qualsiasi reddito.

Anche in questo caso la selezione degli input non è stata casuale, bensì abbiamo prodotto delle combinazioni che permettano di effettuare una *coverage* del codice quanto più vicina al 100%. Per fare ciò è necessario inserire dei parametri che consentano di ricoprire ciascun *branch* di esecuzione.

In questo caso non è stato complesso perché è stato sufficiente inserire cinque redditi che appartenessero ciascuno ad uno scaglione diverso. Facendo ciò, abbiamo ottenuto una coverage del 100% sulla procedura *irpef_ calculation(income)*.

Non essendo invece rilevanti i test sul *main* (verrà sempre eseguito all'avvio del programma), abbiamo inserito la clausola ♯*pragma: no cover* che ci consente di escludere la porzione di codice scelta dall'analisi del programma.

Per verificare la correttezza dei test presenti nel file *test.py* è sufficiente avviare il Bash-Script chiamato *script.sh* che eseguirà tutti gli *assertEqual* presenti e riporterà i risultati della coverage nel file *report.txt*, mostrando (come nell'esercizio precedente) le seguenti informazioni relative a *irpef.py*:

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

Come nell'esercizio precedente, lo script riporterà anche il risultato dei test in formato *HTML*. In esso sarà riportato il codice sorgente del programma del quale si potranno evidenziare quattro tipi di informazione attraverso i bottoni presenti nella parte superiore:

- statement eseguiti (*run*) in verde;
- statement non eseguiti (*missing*) in rosso;
- statement esclusi dall'analisi (*excluded*) in grigio;
- statement parzialmente eseguiti (*partial*) in giallo (ad esempio costrutti *if* privi del ramo *else*).

Esercizio 3 - Calcolo della Pasqua

Codice dell'algoritmo

Per lo svolgimento di questo esercizio ci siamo basati sull'algoritmo di *Oudin-Tondering* per il calcolo della *Pasqua*, presente a questo indirizzo: `http://calendario.eugeniosongia.com/oudin.htm`.

Il file *easter.py* deve essere eseguito ricevendo in input un solo parametro: l'anno del quale si vuole calcolare il giorno della Pasqua. Questo parametro, dovrà essere un numero maggiore di 1582, anno in cui è stato introdotto il nostro *calendario Gregoriano* e successivamente verrà passato come parametro alla seguente procedura:

- *easter_ day(year)*: questa funzione si occuperà di applicare l'algoritmo sopraindicato per individuare il mese e il giorno esatti della Pasqua per l'anno indicato dall'utente.

  Successivamente si creano diversi branch per migliorare la stampa a video del risultato:

  - il mese verrà convertito da numero a stringa di letterali;
  - il giorno verrà seguito dai suffissi *st*, *nd*, *rd*, *th* a seconda della sua ultima cifra.

  Infine la procedura ritornerà una stringa contente il risultato del calcolo che verrà stampata a video nella funzione *main* del programma.

Testing

In seguito abbiamo creato il file *test.py* nel quale sono contenuti gli *unit test* relativi alla procedura sopradescritta. Le parti principali di questi test sono gli *assert* e, anche in questo caso, ci siamo orientati sull'utilizzo degli *assertEqual* per verifica che la procedura, con eventuali input, restituisca il risultato atteso.

Questi test sono stati utili per verificare il corretto funzionamento dell'algoritmo implementato, infatti all'indirizzo `http://digilander.libero.it/acqua67/calcolo%20della%20data%20di%20pasqua.htm` è possibile effettuare online il calcolo della Pasqua per qualsiasi anno.

Anche in questo caso la selezione degli input non è stata casuale, bensì abbiamo prodotto delle combinazioni che permettano di effettuare una *coverage* del codice quanto più vicina al 100%. Per fare ciò è necessario inserire dei parametri che consentano di ricoprire ciascun *branch* di esecuzione.

È stato sufficiente inserire quattro anni in cui la Pasqua arrivasse nei mesi di Marzo e Aprile e in giorni in cui la ultima cifra fosse 1, 2, 3 e qualsiasi altra. Facendo ciò, abbiamo ottenuto una coverage del 100% sulla procedura *easter_ day(year)*.

Non essendo invece rilevanti i test sul *main* (verrà sempre eseguito all'avvio del programma), abbiamo inserito la clausola ♯*pragma: no cover* che ci consente di escludere la porzione di codice scelta dall'analisi del programma.

Per verificare la correttezza dei test presenti nel file *test.py* è sufficiente avviare il Bash-Script chiamato *script.sh* che eseguirà tutti gli *assertEqual* presenti e riporterà i risultati della coverage nel file *report.txt*, mostrando (come negli esercizi precedenti) le seguenti informazioni relative a *easter.py*:

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

Come negli esercizi precedenti, lo script riporterà anche il risultato dei test in formato *HTML*. In esso sarà riportato il codice sorgente del programma del quale si potranno evidenziare quattro tipi di informazione attraverso i bottoni presenti nella parte superiore:

- statement eseguiti (*run*) in verde;
- statement non eseguiti (*missing*) in rosso;
- statement esclusi dall'analisi (*excluded*) in grigio;
- statement parzialmente eseguiti (*partial*) in giallo (ad esempio costrutti *if* privi del ramo *else*).

Esercizio 4 - Monitoraggio dei processi

Il seguente esercizio si basa principalmente sull'utilizzo del comando bash *strace*. Esso è un tool utile per la diagnostica e il debugging. Permette di tracciare e salvare le chiamate di sistema di un processo.

Nel caso più semplice, *strace* esegue il comando indicato. Intercetta e salva le chiamate di sistema che sono chiamate dal processo e i segnali da esso ricevuti. Il nome di ogni chiamata, i suoi argomenti ed il suo valore di ritorno vengono stampati sullo *standard error* o sul file specificato tramite l'opzione *-o output_filename*.

strace può essere arricchito con delle opzioni che andranno a modificare il suo output secondo alcuni parametri. Nel nostro codice sono presenti le seguenti opzioni:

- *-o output_filename*: riporta l'output del comando nel file indicato;
- *-ff*: se è presente l'opzione *-o output_filename*, ogni processo tracciato viene scritto in *filename.pid* (dove *pid* è il codice identificativo del processo);
- *-p pid*: permette di specificare il pid del processo che si vuole monitorare con *strace*.

Codice

Per lo svolgimento di questo esercizio è stata implementata una procedura nella quale un processo padre genera un figlio e scambia messaggi con esso.

Nel file C *fork.c* il processo padre, attraverso la chiamata di sistema *fork()*, genera un processo figlio. Successivamente il padre scriverà un messaggio sulla *pipe* che verrà letto dal figlio e stampato a video.

Lo script *monitor.sh* esegue il comando *strace* sul processo *run* (l'eseguibile del file *fork.c*) e ne salva il risultato in due file di testo chiamati *dump.txt.pid*.

Dai nomi dei file verranno estratti i *pid* dei processi per individuare il processo padre (quello con *pid* minore). Si analizza il corrispondente file *dump* per verificare se ha eseguito qualche *fork()* (nell'output di *strace* è indicata come *clone*) e scrive nel file *forks.log* il numero di processi creati ed il pid di ogni figlio.

Successivamente vengono analizzate anche le chiamate di sistema *write(fd, str, len(str))* del processo padre. Da queste chiamate si estrae il *file descriptor* e si controlla che sia maggiore di 2, poiché:

- lo standard input ha 0 come *file descriptor*;

- lo standard output ha 1 come *file descriptor*;
- lo standard error ha 2 come *file descriptor*;
- la pipe creata dal processo padre ha un *file descriptor* maggiore di 2.

Se vengono trovate *write(fd, str, len(str))* con $fd \geq 3$, si salvano i messaggi scambiati sul file *comm.log*.

Analisi di una sessione ssh

Sono inoltre presenti due script bash per verificare la possibilità di furto di dati sensibili, quali credenziali di accesso, sul processo *ssh*.

Il file *ssh-start.sh* avvia una sessione *ssh* collegandosi a localhost. La sessione viene poi chiusa con *exit* per consentire il completamento del processo di intercettazione delle chiamate di sistema.

Il file *ssh-sniffing.sh* recupera il pid del processo *ssh* da monitorare ed esegue il comando *strace* su di esso generando il file *ssh.log*. Lo script chiede poi la password all'utente per fornire una prova del fatto che questa sia stata intercettata.

Esercizio 5a (semplificato) - Monitoraggio degli accessi a 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.

ltrace può essere arricchito con delle opzioni che andranno a modificare il suo output secondo alcuni parametri. Nel nostro codice sono presenti 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*.

Codice

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.

Come suggerito abbiamo sfruttato la subdirectory */var* della root directory in Linux, la quale contiene i file che vengono scritti dal sistema durante il corso delle sue operazioni.

Nel file C *illegal_folder_access.c* si tenta di creare e aprire un file *test.txt* nella directory */var* con permessi di scrittura e successivamente si vuole scrivere un carattere in esso.

La funzione di libreria *fopen("/var/test.txt", "w")* ritornerà il valore *NULL* perché il processo non detiene i permessi per la creazione dei file in tale directory.

Lo script *monitor.sh* crea un file di testo temporaneo (*ltrace_log.txt*) in cui salvare l'output del comando *ltrace* e il file *log.txt*.

Dopo aver eseguito il comando *ltrace* per tracciare le chiamate dinamiche alle librerie (eseguite dal processo in esecuzione) si filtra l'output per ottenere solamente i record che fanno riferimento alla funzione di libreria (*fopen*).

Fatto ciò, si scorrono tutti i record rimanenti e per ciascuno di questi si estrae il percorso a cui fa rifermento la funzione. Da questo sarà poi possibile ricavare il proprietario della directory alla quale si vuole accedere.

Confrontando l'utente con il dato ricavato si è in grado di affermare se si tratta di un accesso regolare o meno e ciò verrà riportato nel file *log.txt*.