*Luca Comparini S4184885*

LABORATORIO 2: IncApache

In questo laboratorio si vuole creare un web server in grado di rispondere a richieste HTTP 1.0 tramite protocollo TCP; I metodi implementati sono GET ed HEAD. La dinamica di comportamento del serve(HTTP 1.0) è quella di chiudere le connessioni con i client dopo aver dato un risposta, le connessioni sono non permanenti.

Il server usa il meccanismo dei thread: all’interno del server vengono creati diversi thread, ciascuno dei quali viene poi dedicato alla gestione di una delle connessioni possibili richieste dai client. Questa soluzione comporta il rischio di problematiche di race condition e di bug come i deadlock. Le race condition vengono evitate con l’uso di una struttura di sincronizzazione chiamata mutex.

Il server implementa l’uso di rootjail a scopo di sicurezza, infatti queste impediscono a utenti malevoli di accedere a directory che potrebbero essere private oppure potrebbero permettere di modificare il comportamento del server stesso. Le rootjail si implementano tramite la system call chroot(), questa richiede i privilegi di amministrazione, condizione che dovrà poi essere cambiata tramite la funzione setuid() in modo da impedire che successivamente si possa eludere la rootjail.

Il programma viene eseguito con privilegi di amministratore poi si divide in due processi tramite una fork, questi sono collegati tramite pipe. Uno di essi chiamerà chroot() per poi perdere i privilegi di amministratore mentre l’altro manterrà l’accesso a /bin in modo da poter chiamare il comando file (con opzione -i) necessario per leggere il MIME type dei messaggi.

Passando al codice, i file da completare sono:

* incApache \_main.c
* incApache\_http.c
* incApache\_threads.c
* incApache\_aux.c

Le funzionalità da implementare nel server sono il conditional get ed il meccanismo dei cookie tramite header opzionali.

Il conditional get si usa per evitare di dover inviare la risorsa richiesta dal client ad ogni evenienza, infatti il client può mandare l’header opzionale if\_modified\_since seguito da una data. Il serve, nel qual caso, confronta la data con quella descritta nell’inode corrispondente a quel file; per accedere all’inode il server usa la funzione stat(). Se la risorsa richiesta non è stata più modificata dall’ultima volta che il client l’ha ottenuta allora si procede alla risposta 304 not modified senza “allegare” la risorsa richiesta.

Il meccanismo dei cookie: quando un client si connette per la prima volta (o meglio senza cookie) il server provvede a generare un cookie con una data di scadenza ed un identificatore (nel caso del laboratorio corrisponde all’indice di un array la cui cella tiene conto del numero di richieste fatte con quell’identificatore) e risponde con setcookie inoltrando il cookie che ha creato. Nelle successive comunicazione il client può farsi riconoscere tramite il cookie dato dal server e, nel caso, il serve terrà traccia del numero di richieste di quel client.

Per quanto riguarda la parte di debugging dopo essere arrivato ad avere un server apparentemente funzionante ho iniziato una fase preliminare di controllo delle risposte tramite telnet. Riporto alcuni esempi di comportamento adeguato del server:

Immagine che contiene testo

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

Purtroppo in questa fase di sperimentazione tramite telnet non sono riuscito a implementare manualmente i cookie e quindi ad avere prova del loro comportamento.

Successivamente mi sono posto il problema di cercare di porre sotto stress il server creato per far in modo di evidenziare eventuali dinamiche di race condition e per provare la resilienza del programma. Infatti in un programma multi thread si possono verificare errori quando più thread sono attivi (errori che potrebbero non manifestarsi altrimenti) quindi per essere ragionevolmente convinti del funzionamento del programma è necessario avere più richieste in una quantità di tempo molto piccola.

Per fare ciò ho scelto di usare Jmeter, impostandolo in modo da avere più richieste contemporaneamente e anche per file diversi del server.

Di seguito lascio il resoconto di una prova di stress tramite Jmeter con 2 gruppi di 100 thread per richieste HTTP di tipo GET, i due gruppi di thread richiedono due file diversi.

Immagine che contiene testo, screenshot, elettronico

Descrizione generata automaticamente

Immagine che contiene testo, screenshot, elettronico

Descrizione generata automaticamente