UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA MAGISTRALE IN TECNOLOGIE DELL'INFORMAZIONE
E DELLA COMUNICAZIONE

RACE CONDITION IN APPLICAZIONI WEB

Relatore: Prof. Danilo BRUSCHI
Correlatore: Dott. Roberto PALEARI

Tesi di Laurea di:
Davide MARRONE
Matricola 685167

Anno Accademico 2009–10

Ai miei genitori

Ringraziamenti

Mi sembra doveroso iniziare la tesi ringraziando le tante persone che mi hanno supportato in questi ultimi anni.

Prima di tutto ringrazio il Prof. Danilo Bruschi e il Dott. Roberto Paleari per avermi dato la possibilità di svolgere questa tesi, per avermi dato una *grande* mano nel realizzarla e per avermi sostenuto nel portare avanti le mie idee. Ringrazio il Dott. Mattia Monga per essersi interessato al mio lavoro e per gli utili suggerimenti ed il tempo dedicatomi.

Un grazie di cuore ai miei genitori, per avermi permesso di arrivare fino a questo traguardo, per avermi costantemente seguito nelle mie scelte consentendomi ogni volta di compierle con la massima libertà e per avermi sempre sostenuto nei momenti più duri aiutandomi ad andare avanti. Anche se non ve lo dico mai abbastanza grazie per tutto. Ringrazio mio fratello Alberto, la nonna Nilde e lo zio Mario per essermi sempre vicini.

Un ringraziamento speciale a Riccardo “Orte”, Roberto e Luca per tutti i momenti trascorsi insieme in università, grazie per tutti i suggerimenti, i consigli e i confronti che hanno reso gli ultimi anni di lezioni ed esami sicuramente meno pesanti, più piacevoli e anche un po’ divertenti.

Un ringraziamento particolare a tutti gli “hacker” del laboratorio LaSER, durante il tempo trascorso in laboratorio ho imparato in breve tempo una quantità di cose per cui normalmente sono necessari anni di lavoro. Ringrazio (rigorosamente in ordine alfabetico): Alessandro, Andrea, Aristide, Emanuele, Gianpaolo “Gianz”, Lorenzo, Lorenzo “Gigi Sullivan”, Luca, Stefano per la bella accoglienza che mi hanno riservato, per tutti i consigli, le discussioni, i CTF e la fantastica avventura a Las Vegas.

Concludo ringraziando i miei amici più cari, con cui ho condiviso in questi ultimi anni diversi momenti per me importanti, sono sempre stati presenti e mi hanno sostenuto sempre, ringrazio in particolare (rigorosamente in ordine alfabetico): Andrea,

Andrea “Masce”, Francesco “Cecco”, Greta, Laura, Marco “Bazzo”, Sara, Simone. Un ringraziamento speciale va a Francesco “Cecco” che con molta pazienza ha riletto la versione finale di questo lavoro alla ricerca di refusi, errori grammaticali e mi ha suggerito alcune modifiche che hanno reso la tesi più leggibile.

Indice

1 Introduzione **1**

1.1 Obiettivi del lavoro 4

1.2 Organizzazione della tesi 5

2 Concetti preliminari **6**

2.1 Applicazioni web 6

2.2 Sicurezza nelle applicazioni web 8

2.2.1 Sessioni 9

2.2.2 Cross Site Scripting 11

2.2.3 Cross Site Request Forgery 14

2.2.4 Command-Injection 16

2.2.5 SQL-Injection 18

2.3 Race condition 21

3 Scenario **25**

3.1 Concorrenza nelle applicazioni web 25

3.2 Analisi del problema 27

4 Rilevamento di race condition in applicazioni web **30**

4.1 Il metodo di rilevamento 30

4.2 Analisi dettagliata dei componenti 31

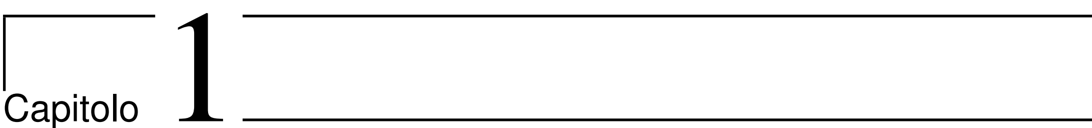
4.2.1 Logger di query SQL 31
4.2.2 Analizzatore off-line: approccio semplice 32
4.2.3 Analizzatore off-line: riduzione dei falsi positivi 35
4.2.3.1 Clausole WHERE 35
4.2.3.2 Associazione degli attributi alle tabelle 37
4.2.3.3 Annotazioni 37
4.3 Implementazione 39
4.4 Criticità dell'architettura 40
4.5 Valutazione del prototipo 42
4.6 Contromisure 44

5 Race condition in applicazioni web reali 47
5.1 Individuazione di race in software closed-source 47
5.2 Caratteristiche della vulnerabilità 49
5.3 Exploiting 50

6 Lavori correlati 54
6.1 Analisi statica 54
6.2 Analisi dinamica 55
6.3 Model checking 55
6.4 Classificazione della soluzione proposta 56

7 Conclusioni 57

Bibliografia 59



Introduzione

La maggior parte delle moderne applicazioni sono sviluppate utilizzando il paradigma web, il modello adoperato è di tipo client-server e viene impiegato il protocollo *HTTP* per lo scambio dei dati. In questo modello client e server sono rappresentati rispettivamente da un browser web ed un server web, quest'ultimo solitamente fornito di moduli che permettono l'esecuzione di codice lato server. Le applicazioni che presentano tali caratteristiche sono generalmente chiamate "*applicazioni web*".

Originariamente queste applicazioni erano realizzate utilizzando un semplice meccanismo che consentiva la creazione di pagine web. Una delle prime tecnologie realizzate è stata la Common Gateway Interface (CGI) [25], progettata con lo scopo di fornire un accesso di tipo web ad applicazioni legacy creando un gateway tra il server web e le applicazioni esistenti. Questo tipo di piattaforme non sono più utilizzate.

Oggi la maggior parte degli approcci impiegati, per la realizzazione di applicazioni web, consistono nell'estendere il server web con moduli in grado di offrire ai programmatori dei framework molto semplici e flessibili per lo sviluppo di applicazioni. In genere il server web permette di inizializzare una virtual machine che è in grado di interpretare il codice delle applicazioni, tipicamente scritte con linguaggi con tipizzazione dinamica come PHP, Python, Ruby o ASP.NET.

Generalmente le applicazioni web fanno affidamento su un'architettura a tre strati basata su un browser web, un server web ed un database manager. Una piattaforma ampiamente utilizzata, è quella LAMP [22] che è formata da una macchina con sistema

operativo **L**inux su cui è installato un server web **A**pache in grado di gestire **M**ySQL come database management system attraverso degli script creati con il linguaggio **P**HP.

Negli ultimi anni le applicazioni web sono state oggetto di differenti attacchi, la maggior parte dei quali specifici all'ambiente in cui vengono eseguite [11]; tipicamente queste vulnerabilità possono portare alla compromissione di informazioni sensibili. Mentre le applicazioni tradizionali sono prevalentemente vulnerabili a errori basati sulla memoria (es., buffer overflow, format bug), le applicazioni web sono realizzate con framework che impediscono questi tipi di attacchi. Queste sono però vulnerabili ad altri attacchi legati ai componenti necessari per il loro funzionamento (es., DBMS, browser web). Secondo un'analisi condotta negli ultimi anni [37], più del 60% delle vulnerabilità software riportate annualmente sono relative ad applicazioni web. Questo fenomeno è dovuto principalmente al fatto che è abbastanza semplice creare un'applicazione web, molte applicazioni sono scritte da sviluppatori con poca esperienza di programmazione e di sicurezza. Inoltre le applicazioni web rappresentano obiettivi interessanti perché spesso sono un'interfaccia verso i server di back-end che gestiscono informazioni sensibili (es., numeri delle carte di credito, dati finanziari, indirizzi e-mail). La maggior parte dei difetti presenti nelle applicazioni web sono dovuti all'interazione tra l'applicazione e il database management system usato come meccanismo di storage di lungo termine [17], altri dipendono da una incorretta gestione della fiducia tra client e server [6]. Questi tipi di vulnerabilità possono essere attribuite ad una mancata o incorretta validazione dell'input fornito dall'utente: alcuni dati che sono inseriti dal client non sono opportunamente verificati.

In questo lavoro di tesi viene introdotta e discussa una nuova tipologia di vulnerabilità che riguarda le applicazioni web. Questa vulnerabilità è emersa osservando il comportamento di alcune applicazioni sottoposte ad un alto carico di lavoro con molte richieste contemporanee. Si è osservato che anche le applicazioni web soffrono del tipico sintomo delle race condition.

Il problema della concorrenza è ben noto nel campo della sicurezza informatica, in questa tesi viene mostrato che l'impatto di questo problema sulle applicazioni web non è stato sufficientemente esplorato. Generalmente le applicazioni web sono composte da diversi script, ognuno dei quali compie un'azione semplice e ben definita, facil-

mente descritta con codice sequenziale. Tipicamente i programmatori web realizzano questi script immaginando che siano eseguiti dall'inizio alla fine senza essere interrotti. Questa assunzione è ovviamente sbagliata: non bisogna dimenticare che quando un utente richiede l'esecuzione di uno script lato server, questo diventa il corpo di un nuovo thread o di un nuovo processo che viene eseguito in un ambiente multi-thread o multi-processo. Questo meccanismo implica che più istanze dei vari script, di cui è composta l'applicazione, possano essere eseguite concorrentemente all'interno di un singolo sistema. Se gli script sono concepiti come semplici programmi composti da codice sequenziale e utilizzano risorse condivise (es., una base di dati o il filesystem), l'esecuzione parallela di istanze multiple di questi script può provocare problemi di concorrenza.

Nonostante le race condition siano una vulnerabilità nota, la tesi mostra che i programmatori web trascurano questo aspetto di sicurezza. Spesso non viene considerato il fatto che le applicazioni utilizzano delle risorse condivise in un ambiente concorrente e di conseguenza devono essere impiegato degli appropriati meccanismi di sincronizzazione. Partendo da questa considerazione si è verificato che, in alcuni casi, l'esecuzione parallela di diversi script o diverse istanze dello stesso script, concepiti come sequenziali, può portare a percorrere dei cammini non previsti, permettendo ad un utente maligno di alterare il comportamento del programma. Negli esperimenti svolti, sfruttando questo problema di sicurezza, è stato possibile oltrepassare le protezioni contro il brute forcing, alterare il funzionamento di gateway di SMS, aggirare i filtri anti-flood ed infine votare più volte in alcuni sondaggi dove era teoricamente possibile esprimere una sola preferenza.

Bisogna osservare che questa vulnerabilità è molto differente rispetto a quelle tipicamente presenti nelle applicazioni web. Infatti, mentre le SQL injection [17], Command Injection [36] e Cross Site Scripting [6] nascono a causa di una incorretta sanitizzazione dell'input, una race condition è relativa ad un problema di concorrenza: alcune interazioni non previste tra differenti istanze dell'applicazione originale modificano il comportamento del programma rispetto a come era stato ideato dal programmatore.

In particolare, in questo lavoro si è individuata una strategia valida per il rilevamento di una particolare classe di race condition, quelle che possono nascere a causa

dell'utilizzo scorretto, da parte dell'applicazione, di un database relazionale. Il problema di rilevare ed eliminare le race condition è stato ampiamente discusso nella letteratura [26, 18], ma tutti i lavori svolti si focalizzano su applicazioni concorrenti tradizionali. Il problema analizzato in questo lavoro è differente in quanto legato al rilevamento di problemi di sincronizzazione in codice pensato in maniera sequenziale ed eseguito in un ambiente concorrente. Il problema non consiste, come al solito, nell'analizzare il corretto uso delle primitive di sincronizzazione da parte di un programmatore, ma di identificare le situazioni in cui l'esecuzione di un pezzo di codice sequenziale può portare ad un problema di sicurezza, quando più istanze dello stesso codice sono eseguite in modo concorrente.

1.1 Obiettivi del lavoro

Lo scopo di questo lavoro di tesi consiste nel:

- Fare luce sull'impatto delle race condition nelle applicazioni web. Nonostante questo problema di sicurezza sia ben noto, gli effetti di questa vulnerabilità nell'ambito web non sono stati esplorati a fondo.
- Proporre una tecnica per l'individuazione delle race condition causate da una incorretta gestione, da parte dell'applicazione web, del database management system. Il metodo proposto è stato implementato in un prototipo che ha portato alla scoperta di diverse vulnerabilità sconosciute in software open-source ampiamente utilizzati.
- Mostrare alcune possibili contromisure per impedire ogni eventuale tentativo di attacco.
- Mostrare che è possibile realizzare con successo un attacco su di un'applicazione vulnerabile sfruttando alcune particolarità del protocollo *HTTP*.

1.2 Organizzazione della tesi

Questo lavoro di tesi è strutturato nel seguente modo: nel Capitolo 2 sono descritte le principali vulnerabilità tipicamente presenti nelle applicazioni web e viene introdotto il problema delle race condition. Nel Capitolo 3 viene illustrato come questo problema di sincronizzazione si presenta in ambiente web e vengono forniti i dettagli di una race presente in una possibile applicazione di trasferimento credito. Il Capitolo 4 introduce la strategia utilizzata per la rilevazione di race, presenta alcuni dettagli dell'implementazione di un prototipo sviluppato a questo scopo e riporta alcuni risultati sperimentali. Il Capito 5 analizza l'impatto che questo problema di sincronizzazione ha sulle applicazioni reali, si discutono delle possibili tecniche per portare a termine un attacco in ambiente web evidenziando la strategia migliore. Nel Capitolo 6 sono analizzati alcuni lavori che trattano il problema delle race condition in applicazioni tradizionali come termine di paragone per il lavoro svolto. Infine, il Capitolo 7 riassume i principali risultati ottenuti in questo lavoro.

Capitolo 2

Concetti preliminari

Il presente capitolo si apre con l'introduzione ai concetti fondamentali relativi alle applicazioni web e alla loro sicurezza. Vengono poi discusse le principali minacce che possono essere presenti nelle applicazioni web ed infine viene introdotto il problema delle race condition.

2.1 Applicazioni web

Nato come sistema per distribuire su Internet documenti o elementi multimediali statici collegati tra di loro, il web, negli anni, si è trasformato in un'infrastruttura complessa sulla quale è possibile realizzare applicazioni distribuite. Questa trasformazione è avvenuta grazie alla creazione di componenti software necessari per l'esecuzione di applicazioni complesse. Ciò ha inevitabilmente introdotto delle problematiche relative alla sicurezza. Oggi in rete sono presenti milioni di server, che ospitano applicazioni web, in attesa di richieste da parte di client; tutto è basato sul paradigma client/server nel quale le comunicazioni sono incapsulate all'interno del protocollo *HTTP*. Il successo del web è dovuto anche alla semplicità con cui è possibile realizzare un'applicazione: chiunque abbia delle conoscenze basilari di programmazione è in grado di realizzarne una. L'ambiente web presenta infatti diversi vantaggi che semplificano notevolmente lo sviluppo: la parte di rendering dell'applicazione è delegata totalmente al browser web, il programmatore deve solo definire, attraverso un linguaggio di

markup, quali componenti inserire nella sua applicazione; la parte di concorrenza è totalmente nascosta al programmatore che si deve occupare di scrivere il codice come se fosse un semplice programma sequenziale, spetterà successivamente al server web o al framework sottostante far sì che l'applicazione possa essere utilizzata contemporaneamente da più utenti. Infine il protocollo *HTTP*, insieme al paradigma client/server, permette di avere un'applicazione distribuita nella quale i dati sono tutti archiviati in un unico punto e gli utenti possono utilizzare l'applicazione da qualsiasi PC dotato di un browser web.

Per la realizzazione delle moderne applicazioni web sono state sviluppate, da diversi produttori, molte tecnologie che permettono di semplificare il processo di sviluppo e di rendere le applicazioni sempre migliori sotto vari aspetti: grafica, usabilità, prestazioni, ecc. Si possono raggruppare tutte le tecnologie sviluppate in due grandi categorie: tecnologie *lato client* e tecnologie *lato server*. La differenza fondamentale tra queste due categorie riguarda la locazione presso la quale è eseguito il codice dell'applicazione. Per le tecnologie lato client, il codice è eseguito sulla macchina dell'utente, tipicamente grazie alla presenza di appositi moduli all'interno del browser web (es., JavaScript, VBScript) oppure tramite un'apposita virtual machine per l'esecuzione di codice binario scaricato dal server (es., Flash, Java applets). Nelle tecnologie *lato server*, il codice viene eseguito sul server; in questo caso le tecniche maggiormente utilizzate consistono nell'aggiungere al server web i moduli necessari per l'interpretazione del codice, oppure nell'interfacciare il server web a processi indipendenti che possono anche risiedere su macchine differenti.

Per la realizzazione di applicazioni web, dalla loro nascita ad oggi, sono state sviluppate ed adottate moltissime tecnologie, sia lato client che lato server. Le tecnologie lato client servono tipicamente a migliorare un'applicazione per quanto riguarda la grafica e l'usabilità, mentre quelle lato server sono fondamentali per il loro funzionamento. Nel corso degli anni sono state sviluppate soluzioni lato server sia in ambito open-source che closed-source. Oggi le tecnologie maggiormente utilizzate fanno uso tipicamente di linguaggi dinamici orientati agli oggetti (es., PHP, Python, Ruby o ASP.NET).

2.2 Sicurezza nelle applicazioni web

La semplicità con cui è possibile realizzare un'applicazione web ha permesso ad un numero molto elevato di programmatori di sviluppare in questo ambiente; grazie a questa semplicità sono state realizzate applicazioni da programmatori che non hanno forti competenze di programmazione e tanto meno di sicurezza. Non bisogna infatti dimenticare che i componenti coinvolti sono tutt'altro che banali e, se non si conoscono a fondo tutte le problematiche relative a questo ambiente, è abbastanza facile commettere degli errori relativi alla sicurezza, compromettendo così l'intera applicazione e i dati che utilizza.

L'esplosione del web ha portato moltissime organizzazioni, private e pubbliche, a realizzare un proprio sito o un'applicazione web per offrire una serie di servizi ai propri utenti. La tipologia di queste applicazioni è estremamente eterogenea e alcune di queste lavorano su dati importanti e sensibili. Se lo sviluppo di tali prodotti è stato affidato a programmatori inesperti è possibile che la sicurezza dell'applicazione e del sistema che la ospita possa essere compromessa.

I problemi di sicurezza in ambiente web sono legati alle tecnologie utilizzate per la realizzazione delle applicazioni, in particolare, esistono delle vulnerabilità che sfruttano il protocollo di comunicazione, altre che sfruttano le tecnologie lato client ed infine esistono vulnerabilità che si ripercuotono sulle tecnologie lato server e sui componenti collegati ad esse, come ad esempio i DBMS.

La tipologia di attacchi che possono essere eseguiti su un'applicazione web sono molteplici, possono essere relativi all'autenticazione, all'autorizzazione, all'accesso ad informazioni presenti sul client, all'esecuzione automatica di azioni sul client, ecc. Uno degli attacchi più diffusi è sicuramente quello legato all'autenticazione: un utente maligno riesce ad ottenere un accesso all'applicazione senza essere in possesso di credenziali valide. Questo tipo di attacco può essere portato a termine utilizzando diverse tecniche, di seguito viene discusso come sia possibile realizzarlo sfruttando le sessioni, utilizzando una vulnerabilità di tipo XSS [6] o di tipo SQLI [17].

2.2.1 Sessioni

Come già detto in precedenza, il web utilizza il protocollo *HTTP* per lo scambio di dati tra client e server. *HTTP* è un protocollo *state-less*: ogni richiesta è indipendente dalle altre, anche se sono effettuate dallo stesso client. Questa scelta di progettazione, sicuramente valida quando il protocollo è stato ideato, risulta insufficiente per applicazioni web dinamiche. La maggior parte delle applicazioni ha infatti la necessità di tenere traccia delle informazioni relative ad ogni utente. Per queste applicazioni è stato necessario creare dei meccanismi allo scopo di memorizzare lo stato del programma tra le varie richieste effettuate dai client. Oggi tutti i moderni framework di sviluppo offrono ai programmatori delle funzioni che si occupano della gestione delle sessioni. L'utilizzo delle sessioni comporta che i dati relativi alla sessione siano trasmessi al server ad ogni richiesta che viene effettuata. Dato che la comunicazione avviene attraverso il protocollo *HTTP*, anche le informazioni relative alle sessioni devono essere inserite in questo protocollo. La maggior parte delle implementazioni delle sessioni fanno affidamento su un identificativo univoco al quale il server associa le informazioni della sessione; questo identificativo viene spedito al client che lo ritrasmette ad ogni richiesta successiva, solitamente utilizzando un cookie. I browser web infatti memorizzano localmente tutte le informazioni di autenticazione (es., i cookie di sessione, le credenziali *HTTP*, le credenziali di domino) e le inviano in ogni richiesta che viene effettuata dall'utente. Questo meccanismo è utilizzato dalle applicazioni per tenere traccia dello stato dell'utente e può essere impiegato anche quando l'utente si autentica nell'applicazione. Dopo aver eseguito il login l'applicazione memorizza le informazioni relative ai privilegi dell'utente, le associa ad un identificativo ed invia questo token al browser utilizzando un cookie. Il browser riceve l'identificativo di sessione, lo salva localmente insieme agli altri cookie e successivamente, quando l'utente effettua un'altra richiesta, inserisce automaticamente il token tra i cookie della richiesta *HTTP*. Questo semplice meccanismo permette di preservare lo stato dell'applicazione, ma introduce un punto critico all'interno della stessa: se le sessioni sono utilizzate per memorizzare i privilegi di un utente, gli *id* di sessione diventano un elemento molto interessante per un utente maligno, in quanto, se riuscisse ad impossessarsene, potrebbe eludere i controlli di

autenticazione ed impersonificare un altro utente del sistema.

Sono possibili diversi attacchi sugli *id* di sessione; dato che questi token sono un elemento critico per la sicurezza di un'applicazione web è buona norma utilizzare le primitive messe a disposizione dal linguaggio di programmazione per la loro gestione. Se un programmatore decidesse di implementare autonomamente le funzioni per la loro gestione, potrebbe ignorare alcuni aspetti di sicurezza ed introdurre una falla nel sistema.

Di seguito sono riportati gli attacchi più comuni relativi agli *id* di sessione:

Intercettazione: se l'attaccante ha la possibilità di intercettare gli *id* analizzando il traffico di rete o, come discusso più avanti, di recuperare l'*id* di una sessione valida con un attacco XSS, può facilmente impersonificare l'utente che ha generato quella sessione. Come contromisura per questo attacco, per quanto riguarda l'intercettazione del traffico di rete, è possibile utilizzare il protocollo SSL/TLS che si occupa di cifrare i dati a livello di trasporto impedendo così ad un utente maligno l'analisi del traffico.

Predizione: uno degli errori di sicurezza che si può commettere nell'implementazione delle funzioni per la gestione delle sessioni consiste nella generazione errata degli *id* di sessione. Bisogna ricordare che gli *id* sono inviati ai client e quindi un attaccante potrebbe, utilizzando l'applicazione web, raccogliere ed analizzare gli *id* generati per dedurre l'algoritmo con cui questi vengono creati. Se la generazione di questi token avviene con algoritmi troppo semplici e deducibili, è possibile per l'attaccante predire la loro sequenza. L'attaccante potrebbe così generare *id* validi, riuscendo ad impersonificare altri utenti.

Brute force: se non è possibile individuare l'algoritmo con cui vengono generati gli *id*, potrebbe essere possibile generare un sottoinsieme abbastanza ampio e riuscire a trovarne qualcuno valido. Generalmente questo attacco non è applicabile se gli *id* hanno una lunghezza minima sufficiente che garantisce un numero elevato di possibili combinazioni: l'individuazione di qualche *id* valido richiederebbe un tempo troppo lungo.

Session fixation: questo tipo di attacco viene portato a termine in più fasi: inizialmente l'attaccante ottiene dall'applicazione un *id* di sessione valido, successivamente, attraverso un attacco di tipo XSS viene fatto utilizzare alla vittima l'*id* di sessione generato dall'attaccante. In queste condizioni la vittima utilizza un *id* che è noto all'attaccante. Se a questo punto effettua delle operazioni sensibili all'interno dell'applicazione, come ad esempio l'autenticazione al sistema, l'attaccante, utilizzando lo stesso *id*, avrà gli stessi suoi privilegi.

Le sessioni sono quindi un elemento molto critico per la sicurezza di un'applicazione web. Oggi i vari framework si occupano della loro gestione e della loro sicurezza nascondendo ai programmatori tutti i dettagli tecnici. Nel caso in cui non sia possibile utilizzare le primitive fornite dal sistema è necessario tenere in considerazione tutti gli aspetti riportati perché, se sottovalutati, risulta abbastanza facile per un attaccante riuscire ad ottenere degli *id* di sessione validi.

2.2.2 Cross Site Scripting

Secondo la classifica periodicamente redatta da OWASP [27] la vulnerabilità maggiormente diffusa nelle applicazioni web è quella di tipo Cross Site Scripting (XSS). Questa vulnerabilità si basa, come molte altre, sulla mancanza di controlli sull'input fornito dall'utente. Sebbene la vulnerabilità risieda nel codice lato server dell'applicazione, questo attacco lavora totalmente sulle tecnologie lato client. L'attacco, dal punto di vista pratico, consiste nel modificare la pagina web, generata dall'applicazione web, aggiungendo del codice *HTML* o del codice *JavaScript*. I Cross Site Scripting sfruttano la fiducia che il client ha nei confronti del server web. Il browser infatti si aspetta che la pagina ricevuta non contenga del codice maligno ma soltanto il codice necessario per il corretto funzionamento dell'applicazione. Se è presente una vulnerabilità di questo tipo, il browser può ricevere dal server web la pagina contenente il codice per l'esecuzione dell'attacco che viene interpretata ed eseguita dal browser senza che l'utente si accorga della presenza del codice inserito dall'attaccante.

Le principali azioni che si possono compiere in presenza di un attacco di questo tipo sono:

- È possibile recuperare i *cookie* del dominio su cui risiede l'applicazione vulnerabile, tipicamente i *cookie* contengono gli *id* di sessione. Se l'attaccante riesce ad accedere a queste informazioni può impersonificare un altro utente come già discusso nella Sezione 2.2.1.
- Modificando il codice *HTML* è possibile intercettare i dati che sono inseriti in qualsiasi form (es., quelli di login) creando un layer trasparente "sopra" al form da intercettare o via *JavaScript* manipolando gli eventi di invio dei dati. Questa tecnica, nota con il nome di *clickjacking*, consente di alterare il normale comportamento delle applicazioni web.
- È possibile far eseguire al browser una serie di richieste *GET* o *POST* in automatico, all'insaputa dell'utente.
- È possibile modificare il contenuto della pagina generata dall'applicazione. Manipolando l'*HTML* via *JavaScript* è infatti possibile modificare o aggiungere qualsiasi elemento alla pagina vulnerabile.

Esistono due categorie di Cross Site Scripting: *reflected* e *stored*. La differenza tra le due tipologie non riguarda la natura dell'attacco, ma la posizione in cui si trova: nel primo caso il codice è inserito nella *URL* della pagina vulnerabile, mentre per i XSS *stored* l'attacco viene memorizzato in maniera persistente all'interno del server web o del DBMS.

Un classico esempio di *XSS* riguarda la funzione di ricerca presente in moltissimi siti web. In questo caso gli utenti hanno la possibilità di inserire delle parole chiave per effettuare una ricerca tra i contenuti presenti nel sito. Per realizzare questa funzionalità, dopo che l'utente ha inserito il testo da cercare, l'applicazione invia la stringa digitata insieme ai parametri della richiesta *GET* alla pagina che visualizzerà i risultati, nella quale è solitamente riportata anche la stringa ricercata dall'utente. Se i dati immessi non sono in alcun modo controllati e su di essi non viene effettuata nessun tipo di sanitizzazione, l'applicazione risulta vulnerabile ad un *XSS*.

Come esempio, si consideri un'applicazione PHP nella quale sia presente uno script per la visualizzazione dei risultati di una ricerca che contenga la seguente istruzione:

```
echo 'Hai cercato: '.$_GET['chiave'];
```

Questo codice è vulnerabile ad un XSS e consente quindi ad un attaccante di manipolare a proprio piacere la pagina che viene generata. L'attaccante potrebbe, ad esempio, far richiedere alla vittima una pagina contente del codice JavaScript per ottenere i suoi cookie, per questo scopo sarebbe sufficiente far inserire nel form di ricerca questo codice *HTML*:

```
<script>document.location('http://evil.it/'+document.cookie)</script>
```

La stringa ricevuta dall'applicazione sarà semplicemente concatenata alle altre parti della pagina e di conseguenza farà parte dell'HTML della pagina generata. In particolare, in questo esempio, il codice aggiunto effettua un reindirizzamento verso un sito sotto il controllo dell'attaccante al quale vengono passati i cookie dell'utente, che potrebbero contenere un *id* di sessione.

Nell'esempio riportato la vittima potrebbe accorgersi facilmente che sta subendo un attacco, si tratta infatti di una semplice dimostrazione di come sia possibile sfruttare la vulnerabilità; in JavaScript si possono realizzare degli attacchi molto più complessi senza che l'utente si possa accorgere di quello che sta avvenendo. Bisogna inoltre ricordare che l'attacco riportato nell'esempio può essere realizzato senza l'utilizzo del carattere *apice*: in PHP è stato introdotto un meccanismo automatico di escaping di alcuni caratteri particolari per mitigare gli attacchi di tipo SQL-injection che impedisce l'esecuzione dell'attacco riportato. JavaScript è però un linguaggio molto potente e flessibile ed è abbastanza semplice realizzare la stessa versione dell'attacco senza l'utilizzo di caratteri particolari come apice o doppio apice.

L'esempio appena discusso appartiene alla categoria di XSS *reflected*: il codice dell'attacco è inserito all'interno della URL. L'indirizzo completo che l'attaccante dovrebbe diffondere sarebbe:

```
http://vulnerabile.it/ricerca.php?chiave=<script>document.location(
'http://evil.it/'+document.cookie)</script>
```

Il codice dell'attacco, dopo essere arrivato al server tramite i parametri passati in *GET*, viene "riflesso" e rispedito alla vittima che ha fatto la richiesta. Nell'esempio, l'attacco è inserito in chiaro nella URL, l'attaccante ha però a disposizione diversi metodi per offuscare il codice, ad esempio può sostituire tutti i caratteri utilizzando delle funzioni di encoding delle URL oppure può utilizzare un servizio di *URL shortening* per sostituire l'URL contenente l'attacco con una apparentemente innocua.

Per quanto riguarda i XSS di tipo *stored*, il codice dell'attacco viene memorizzato all'interno del server web. La procedura per realizzare questo attacco si divide in due fasi. Nella prima, l'attaccante, utilizzando una funzionalità dell'applicazione (es., un messaggio inserito in un forum) invia il codice maligno che deve essere eseguito dalle vittime e l'applicazione vulnerabile salva questo codice in qualche sistema di archiviazione persistente (es., un database oppure un file). Successivamente, quando il client si collega all'applicazione, il server genera la pagina inserendo anche il codice maligno salvato precedentemente. Il browser riceve ed interpreta la pagina e di conseguenza la vittima subisce l'attacco.

Una vulnerabilità di tipo XSS *stored* è sicuramente molto più pericolosa rispetto ad una di tipo *reflected* in quanto, per subire l'attacco, la vittima non deve utilizzare un apposito link generato dall'attaccante ma è sufficiente che utilizzi normalmente l'applicazione. L'effetto di un XSS di tipo *stored* può portare a danni molto importanti. Per fare un esempio, è possibile scrivere dei virus con l'utilizzo di questa tecnica, probabilmente il virus più famoso che è stato realizzato è "Samy" [41] che in 20 ore ha infettato oltre un milione di utenti del social network MySpace.

2.2.3 Cross Site Request Forgery

L'attacco di tipo Cross Site Request Forgery (*CSRF*) consiste nel far eseguire alla vittima una o più richieste decise dall'attaccante sfruttando le credenziali di autenticazione che vengono inserite automaticamente dal browser se la vittima si è autenticata nell'applicazione. Questo attacco sfrutta la fiducia che un sito web ha nei confronti

dei suoi utenti, in quanto un'applicazione web si aspetta che le richieste ricevute da parte di un utente autenticato siano legittime e che siano state effettuate volontariamente dall'utente. Se però l'utente è vittima di un attacco *CSRF*, le richieste inviate al server sono generate dall'attacco stesso e non dall'utente, per l'applicazione web risulta impossibile distinguere le richieste generate dall'attacco rispetto a quelle effettuate dall'utente.

Per capire meglio il funzionamento di questo attacco, di seguito è riportato un semplice esempio. Si supponga di avere un'applicazione generica che tra le impostazioni permetta all'utente di effettuare il cambio dell'indirizzo email con il quale si è registrato. Per realizzare questa funzionalità l'applicazione mette a disposizione un form HTML dove l'utente può inserire il nuovo indirizzo. Quando un utente compila ed invia il form per cambiare il suo indirizzo email, il browser genera una richiesta *HTTP* del tipo:

```
GET /info.php?newEmail=nuovoindirizzo%40dominio.it
Host: www.vuln.it
```

La pagina per la modifica dei dati si chiama `info.php` e riceve come parametro in *GET* il nuovo indirizzo email nella variabile `newEmail`; nell'esempio è stato inserito `nuovoindirizzo@dominio.it` codificato con lo standard urlencode. In questo caso lo scopo dell'attaccante è quello di modificare l'indirizzo email della vittima con un indirizzo in suo possesso, potrà poi utilizzare in un secondo momento la funzione di recupero password ed ottenere o modificare la password della vittima. Per eseguire l'attacco è sufficiente fare in modo che la vittima visualizzi la pagina:

```
http://www.vuln.it/info.php?newEmail=evil%40domain.com
```

Se la vittima accede a questa URL dopo essersi autenticata nell'applicazione, provoca la modifica del suo indirizzo email e di conseguenza l'attaccante riesce a portare a termine il suo attacco. Per far accedere la vittima all'indirizzo indicato l'attaccante può utilizzare delle tecniche di social engineering oppure può creare una pagina web maligna contenente il seguente codice HTML:

```
<img src="http://www.vuln.it/info.php?newEmail=evil%40domain.com"
   width="1" height="1" border="0" />
```

Questo tag può essere inserito in qualsiasi pagina *esterna* all'applicazione, se l'attaccante riesce a far accedere la vittima ad un pagina che contiene quel codice HTML e la vittima è autenticata sull'applicazione vulnerabile, l'indirizzo email verrà modificato. Questo accade perché quando il browser interpreta il codice HTML riconosce il tag dell'immagine e, per recuperarla, effettua una richiesta *GET*, in cui vengono incluse tutte le credenziali che la vittima possiede sul sito vulnerabile. In questo modo l'attacco viene portato a termine senza che la vittima se ne accorga. Lo stesso risultato può essere ottenuto attraverso l'utilizzo di codice JavaScript. Si noti che in questo caso il codice HTML o JavaScript maligno è inserito in una pagina web che non fa parte dell'applicazione vulnerabile ma che può risiedere su qualsiasi server.

Un attacco di tipo *CSRF* in generale costringe l'utente ad eseguire delle azioni senza il suo consenso, un attaccante è in grado di far eseguire alla vittima qualsiasi azione disponibile nell'applicazione web (es., logout, cambio di informazioni personali, acquisto di un prodotto, trasferimento di credito). Per prevenire questo attacco nei normali form HTML le applicazioni web dovrebbero inserire un token in un campo nascosto il cui valore è memorizzato in sessione; successivamente l'applicazione deve verificare che il token presente in sessione sia lo stesso passato nel campo nascosto, in questo modo l'attaccante non può eseguire l'attaccato dato che non può avere accesso al token presente nel form.

2.2.4 Command-Injection

Un'altra delle vulnerabilità più diffuse in ambito web è quella di tipo injection. Anche questa vulnerabilità fa parte di quelle basate sulla mancanza di controlli sull'input e può riguardare diversi componenti utilizzati da un'applicazione web.

Questo difetto si presenta in un'applicazione quando vengono create delle stringhe concatenando una parte statica scritta dal programmatore ed una parte dinamica inserita dall'utente . Un'applicazione è vulnerabile se non controlla i dati inseriti dall'utente ed utilizza le stringhe così generate per effettuare qualche operazione sensibile, come ad esempio l'esecuzione di un comando oppure una query al database. Di seguito vengono analizzate le *shell-injection*, mentre nella successiva sezione sono discusse le

SQL-injection. Esistono anche altre categorie di vulnerabilità di tipo injection che non sono esaminate in questo lavoro: XPath injection [21] e LDAP injection [2].

I linguaggi di programmazione lato server normalmente mettono a disposizione delle funzioni per l'esecuzione di programmi eseguibili presenti sul server, ad esempio: **system**(), **exec**(), **popen**(). Tutte queste funzioni richiedono come argomento una stringa contenente il nome del programma da eseguire seguito da eventuali parametri da utilizzare come argomenti. Un difetto di tipo shell-injection è presente se la stringa, contenente il comando da eseguite, viene generata ed utilizzata senza effettuare nessun tipo di validazione o sanitizzazione sui dati forniti dall'utente. Come esempio si consideri una possibile applicazione per estrarre gli *User-Agent* dai file di log di un server web. Per la realizzazione di questa applicazione potrebbe essere utilizzata l'istruzione:

```
...
system('cut -d \" -f 6 /var/log/apache2/access.log | '
        .' grep -i '.$_GET['chiave']);
...
```

Come si può facilmente notare, i dati forniti dall'utente non sono stati sanitizzati prima di essere utilizzati. In questo caso è molto semplice per l'attaccante eseguire qualsiasi tipo di comando disponibile sul server con gli stessi privilegi in possesso al server web. L'attaccante può, ad esempio, recuperare la lista degli utenti del sistema richiedendo la seguente pagina web:

```
http://www.vuln.it/user_agent.php?chiave=a;%20cat%20/etc/passwd
```

Il carattere *punto e virgola* è considerato dall'interprete dei comandi dei sistemi operativi **NIX* un terminatore, per questo motivo l'attaccante ha la possibilità di poter inserire il nome di un qualsiasi programma che vuole eseguire.

L'esempio di shell-injection mostrato è sicuramente quello più comune che si trova nelle applicazioni che utilizzano funzioni per l'esecuzione di comandi esterni. In alcuni linguaggi di programmazione un attacco simile può essere portato a termine anche in altri modi in base alle funzionalità offerte dal linguaggio stesso. Ad esempio, se l'applicazione utilizza la funzione **eval**() e l'attaccante ha il controllo sul contenuto della stringa passata a questa funzione, è possibile iniettare qualsiasi codice valido che

verrà eseguito dall'interprete del linguaggio. Un'altra possibilità per sfruttare questa vulnerabilità è offerta da alcuni linguaggi di programmazione che permettono l'inclusione di file remoti: se l'attaccante ha il controllo sul prefisso del file che deve essere incluso, potrà inserire come prefisso un indirizzo di una macchina remota che gestisce; quando l'applicazione include questo file, il codice dell'attaccante viene eseguito.

2.2.5 SQL-Injection

La maggior parte di applicazioni web ha la necessità di archiviare e manipolare dei dati. L'utilizzo di semplici file per la memorizzazione di significative quantità di dati risulta spesso insufficiente, per questo motivo la maggior parte di applicazioni web fa uso di un DBMS per l'archiviazione dei propri dati.

Un attacco di tipo SQL-injection si verifica quando viene modificata la logica o la sintassi di una query SQL che viene eseguita da un'applicazione web inserendo nuove keyword SQL o operatori. Le SQL-injection rappresentano una vulnerabilità nota ed ampiamente studiata. In questa sezione vengono introdotti i concetti fondamentali, per approfondire l'argomento si rimanda a [17].

L'idea relativa alle SQL-injection è piuttosto semplice: lo scopo dell'attaccante è sfruttare tutte le sorgenti di input utilizzate dall'applicazione per riuscire a modificare le query eseguite in modo tale da poter alterare il normale comportamento dell'applicazione a proprio piacere. I dati di input che un'applicazione web utilizza possono provenire da diverse "fonti logiche". Come già detto in precedenza, i dati arrivano all'applicazione web attraverso il protocollo *HTTP*, le sorgenti sono quindi tutte quelle legate a questo protocollo. Nel caso in cui non sia possibile accedere al codice sorgente dell'applicazione, il miglior modo per capire quali dati sono utilizzati consiste nell'analizzare il traffico *HTTP* che viene generato durante il suo normale impiego. È abbastanza semplice capire quali sono le variabili passate attraverso richieste *GET* e *POST*, quali sono i cookie utilizzati e quali dati sono inviati ad ogni richiesta. In generale, ogni elemento del protocollo *HTTP* potrebbe essere utilizzato da un'applicazione web per cui anche gli header, come ad esempio lo *User-agent* o il *Referer*, possono far parte delle sorgenti di iniezione. Oltre ai componenti presenti nel protocollo *HTTP*,

c'è un'altra possibile sorgente chiamata *Second Order*. In questo caso, in una prima fase, i dati sono passati attraverso il protocollo *HTTP* e memorizzati all'interno del server in qualche sistema di memorizzazione persistente, successivamente i dati sono recuperati dal sistema di storage ed utilizzati per la costruzione dinamica delle query.

Dopo aver individuato la presenza di vulnerabilità, analizzando tutte le sorgenti di iniezione, è possibile procedere alla realizzazione dell'attacco. Gli obiettivi e la tipologia di attacchi basati su SQL-injection dipendono da molti fattori: dal database utilizzato, da come è stata realizzata l'applicazione web, da come sono gestiti gli errori (es., output a seguito del fallimento di una query SQL). In generale gli obiettivi possono essere classificati in:

Identificazione dei parametri iniettabili: l'attaccante vuole analizzare l'applicazione web e scoprire quali sono tutte le sorgenti d'iniezione.

Database footprinting: l'attaccante vuole scoprire il tipo e la versione del database che l'applicazione web sta utilizzando. Per la realizzazione di questo attacco è necessaria una gestione inadeguata degli errori da parte dell'applicazione.

Determinazione dello schema del database: l'attaccante vuole scoprire lo schema del database, quali sono i nomi delle tabelle, degli attributi e il tipo degli attributi.

Estrazione di dati: l'attaccante vuole estrarre dei dati dal database; questo attacco dipende dall'implementazione dell'applicazione web.

Aggiunta e modifica di dati: l'attaccante vuole inserire o modificare alcuni dati presenti all'interno del database.

Denial of service: l'attaccante vuole impedire l'uso dell'applicazione ad altri utenti, l'attacco può essere portato a termine impostando dei *lock* sulle tabelle o cancellando elementi o dati dal database.

Bypassing dell'autenticazione: l'attaccante vuole eludere il meccanismo di autenticazione realizzato a livello applicativo. L'elusione dell'autenticazione permette all'attaccante di ottenere i privilegi di altri utenti.

Esecuzione remota di comandi: l'attaccante vuole eseguire dei comandi sul server; si tratta di stored procedure o di programmi esterni al database.

A seconda dell'obiettivo, possono essere utilizzate diverse tecniche per la realizzazione dell'attacco. Di seguito viene riportato un semplice esempio che mostra come sia possibile oltrepassare i controlli di autenticazione utilizzando la tecnica delle tautologie.

Una *tautologia* è una condizione logica che risulta sempre vera; si prenda come esempio la seguente riga di uno script PHP:

```
$q = "SELECT id FROM Utente WHERE username = '" . $user . "' "
     ." AND password = '" . $pass . "'";
```

Si assuma che le variabili `$user` e `$pass` siano sotto il controllo diretto dell'attaccante senza essere state in alcun modo sanitizzate: lo scopo di questa query è quello di verificare se nel database sono presenti lo username e la password inseriti dall'utente attraverso un normale form di login. Supponiamo ora che lo scopo dell'attaccante sia quello di eludere l'autenticazione e di effettuare il login con l'utenza di amministratore. Per portare a termine l'attacco è sufficiente impostare le due variabili con i seguenti valori:

```
$user -> admin
$pass -> ' OR '' = '
```

In questo modo la query che viene creata dinamicamente ed eseguita dall'applicazione è:

```
$q = "SELECT id FROM Utente WHERE username = 'admin' "
     ." AND password = '' OR '' = ''";
```

Si può facilmente verificare che la semantica iniziale della query viene totalmente modificata: la valutazione della query dopo la clausola `AND` corrisponde al valore `Vero`. Questo accade a causa della priorità sugli operatori booleani. Nell'esecuzione della query viene prima di tutto valutata la condizione relativa alle due stringhe vuote, questa espressione restituisce come risultato `Vero` e di conseguenza rende superfluo

il controllo dell'attributo password con la stringa vuota presente prima della clausola `OR`. Tutta la parte destra assume il valore `Vero` quindi la query restituisce l'*id* associato alla username *admin*. L'utente maligno può così accedere con i privilegi di amministratore senza essere a conoscenza della sua password.

2.3 Race condition

Dopo aver analizzato le principali problematiche di sicurezza in ambiente web viene ora brevemente esposto il problema delle race condition, il capitolo successivo si concentra esclusivamente sulle race condition in ambiente web.

Una race condition è un comportamento anomalo dovuto ad una dipendenza tra eventi temporali non previsti; si verifica durante l'esecuzione di un programma se il risultato di alcuni passi di computazione dipende dall'ordine con cui lo scheduler imposta l'esecuzione dei singoli thread. Le race condition nascono da un uso improprio di risorse condivise (es., una variabile o una risorsa esterna) da parte di uno o più processi (o thread) che la utilizzano senza impiegare degli appropriati meccanismi di sincronizzazione. Un programma contiene una race condition se due eventi al suo interno sono in conflitto (es., uno legge e l'altro scrive la stessa locazione di memoria) e la loro esecuzione dipende dall'ordine con cui i thread sono eseguiti [18].

Per capire il problema delle race condition è sufficiente analizzare il frammento di codice riportato nella Figura 2.1. Si assuma che x e y siano variabili globali condivise tra i due thread e che non ci sia alcun meccanismo di sincronizzazione. Osservando gli esempi di esecuzione riportati si può facilmente verificare che il risultato finale della variabile y dipende da come lo scheduler organizza l'esecuzione delle singole istruzioni dei due thread. Nel primo esempio di esecuzione, dopo la conclusione dei due thread, il valore di y è 12 mentre nel secondo è 6.

Le race condition sono state ampiamente analizzate in letteratura, Netzer e Miller hanno sviluppato un modello formale per la loro classificazione [26]. Tale modello è composto da due attributi ortogonali: su un asse sono presenti gli attributi *general* e *data* mentre sull'altro asse gli attributi *feasible*, *apparent* e *actual*. Una *data race* viene definita come una coppia di accessi in conflitto che potrebbero essere eseguiti

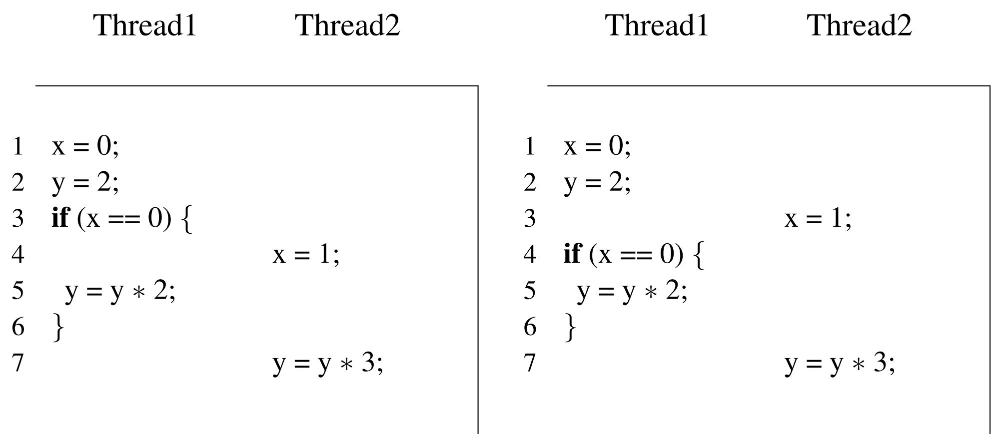


Figura 2.1: Due possibili esecuzioni di un frammento di codice.

contemporaneamente (gli accessi non sono atomici o protetti da qualche tipo di sezione critica); una *general race* è invece composta da accessi in conflitto nei quali l'ordine non è garantito. Una *actual race* si verifica in una particolare esecuzione del programma e si riferisce solo alle *general race*. Una *feasible race*, come suggerisce il nome, è una race che non si verifica nella corrente esecuzione, ma è possibile che si presenti in un'altra. Infine una *apparent race* è una race che apparentemente sembra possibile ma in realtà non può mai verificarsi.

Questo iniziale modello di classificazione è stato ripreso ed ampliato da Helmbold e McDowell [18]. Il loro modello si basa principalmente sull'analisi dell'ordine in cui vengono eseguiti gli eventi all'interno di un programma. Un evento è una sequenza continua di una o più operazioni atomiche eseguite da un singolo thread. Durante l'esecuzione di un programma due eventi possono avere tra di loro diverse relazioni d'ordine in base al momento in cui iniziano e finiscono; ad esempio, possono essere eseguiti sempre in due momenti distinti, a volte l'inizio di uno può verificarsi prima della fine dell'altro oppure la loro esecuzione può avvenire sempre contemporaneamente. Due eventi sono in conflitto se entrambi accedono alla stessa risorsa condivisa e, un evento o entrambi, la modificano. In questo modello, per l'individuazione di race condition viene fissato un input per il programma da analizzare e si osservano diverse esecuzioni. Per ogni istanza è possibile esaminare la relazione d'ordine che si verifica

tra gli eventi dell'applicazione, una race è presente quando ci sono due eventi in conflitto e il loro ordine di esecuzione cambia tra due diverse istanze del programma. In base al tipo di relazione d'ordine in cui gli eventi si verificano è possibile suddividere le race condition in quattro categorie, fissato un input per il programma è presente una race se si verificano queste condizioni:

Concurrent race: in ogni esecuzione del programma si verificano gli eventi `e1` e `e2` e la loro esecuzione avviene concorrentemente.

General race: esistono esecuzioni del programma in cui gli eventi `e1` e `e2` si sovrappongono temporalmente ed esecuzioni in cui `e1` termina prima dell'inizio di `e2` o viceversa `e2` termina prima dell'inizio di `e1`.

Unordered race: esistono esecuzioni del programma in cui `e1` termina prima dell'inizio di `e2` ed esecuzioni in cui `e2` termina prima dell'inizio di `e1` ma in nessuna esecuzione `e1` e `e2` si sovrappongono temporalmente.

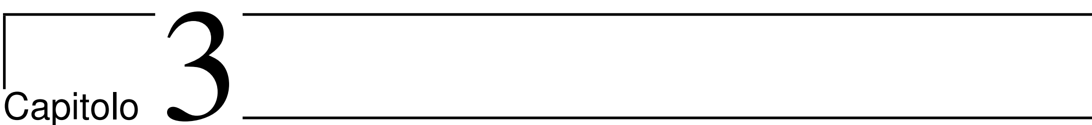
Artifact race: esistono esecuzioni del programma in cui si verifica `e2` ma non `e1` ed esistono esecuzioni in cui `e1` termina prima dell'inizio di `e2` oppure si verifica `e1` ma non `e2`. Non ci sono esecuzioni in cui `e1` e `e2` vengono eseguiti concorrentemente oppure in cui `e2` termina prima dell'inizio di `e1`.

Queste quattro categorie si basano sull'ordine di esecuzione degli eventi su più istanze dello stesso programma con un input fissato. Le race condition possono però avere anche altri importanti attributi per i quali è possibile classificarle, di seguito ne vengono riportati altri tre: come le race influiscono sul flusso di esecuzione del programma, la gravità di una race ed infine la fattibilità di una race.

Controllo vs dati: per definizione in ogni race è presente un conflitto nell'accesso a qualche risorsa condivisa, una caratteristica importante da tenere in considerazione per la loro classificazione riguarda l'influenza delle race sul flusso di esecuzione del programma. Le race possono infatti condizionare il percorso di esecuzione, come ad esempio quella in Figura 2.1, in questo caso sono dette *control race*. Se il flusso di esecuzione non è modificato e il conflitto è solo sui dati sono dette *data race*.

Gravità: le race possono essere classificate per la loro gravità, si possono raggruppare in critiche e benigne. Una race benigna non ha nessun effetto sul risultato finale del programma mentre una race critica può modificarne il risultato. Proteggere una regione critica con meccanismi di mutua esclusione non elimina le race ma fa si che diventino benigne.

Fattibilità: Questa caratteristica non è relativa alle race ma agli strumenti di rilevamento, ogni race che viene riportata dallo strumento ma che in realtà non può mai verificarsi è detta infattibile.



Scenario

In questo capitolo viene analizzato il problema della concorrenza in applicazioni web confrontandolo con quello presente nelle applicazioni tradizionali. Viene poi presentato un esempio di programma vulnerabile che è successivamente richiamato nel corso del lavoro di tesi per illustrare come la soluzione proposta sia in grado di individuarne le vulnerabilità.

3.1 Concorrenza nelle applicazioni web

Una *race condition* si verifica quando più thread o processi, eseguiti concorrentemente, accedono a dati condivisi senza l'utilizzo di appropriati meccanismi di sincronizzazione [26]. Le race condition sono difficili da individuare perché la mente umana non è in grado di immaginare l'altissimo numero di possibilità in cui thread o processi possono essere schedulati ed eseguiti all'interno di un sistema concorrente. Per questo motivo la concorrenza è una tipica fonte di vulnerabilità [3, 4, 12] e uno dei più antichi problemi di sicurezza [1]. La ricerca scientifica, in questo ambito, si è concentrata sulla ricerca di race condition in applicazioni realizzate con diversi linguaggi di programmazione ad alto livello che forniscono delle primitive native di sincronizzazione [35, 34, 10, 14]. Questi studi sono però specializzati su applicazioni che sono state scritte esplicitamente per ambienti multi-thread o multi-processo, in questi casi il

programmatore è ben conscio del fatto che la sua applicazione deve lavorare con più entità che accedono a risorse condivise.

In ambito web il problema della concorrenza è diverso rispetto a quello delle applicazioni tradizionali, nonostante il concetto di base sia lo stesso. Nella maggior parte dei framework oggi utilizzati, per la realizzazione di applicazioni web, gli aspetti di concorrenza sono di fatto nascosti agli sviluppatori. Generalmente non è necessario, e a volte possibile, creare dei thread o processi per l'esecuzione dell'applicazione. I moderni framework infatti si occupano di istanziare ed eseguire concorrentemente i diversi componenti necessari per il corretto funzionamento dell'applicazione. Il lavoro del programmatore viene inoltre semplificato dalla quasi totale assenza di variabili condivise. Solitamente non è infatti possibile creare delle variabili accessibili tra più thread o processi, questo comporta che il programmatore non debba utilizzare delle esplicite primitive di sincronizzazione diminuendo il rischio di potenziali deadlock o problemi di concorrenza. Le uniche variabili condivise sono quelle di sessione ma, in questo caso, il framework garantisce che l'accesso avvenga correttamente in modo esclusivo proteggendole con dei meccanismi di locking.

Le applicazioni web sono tipicamente costituite da un insieme di script indipendenti il cui scopo è quello di realizzare i vari componenti dell'applicazione: dall'interfaccia grafica all'interrogazione e aggiornamento di un database dove vengono salvati i dati. Grazie alle moderne architetture il programmatore dispone di un ambiente nel quale ha l'impressione che ogni script sia un singolo programma che viene eseguito dal server web in maniera sequenziale. In realtà, per l'esecuzione dell'applicazione, il server web o il framework si occupano di istanziare un nuovo thread o processo per ogni nuova richiesta, i quali vengono eseguiti in un ambiente multi-thread o multi-processo.

Sebbene i framework permettano di semplificare molto lo sviluppo di applicazioni, hanno il problema di far trascurare totalmente al programmatore i problemi di concorrenza che si possono verificare. È infatti importante ricordare che i framework nascondono solo una parte dei problemi relativi alla concorrenza. Se però l'applicazione utilizza delle risorse condivise come un DBMS o il filesystem, è compito del programmatore adottare le corrette primitive di sincronizzazione necessarie ad evitare

```
1 $query = mysql_query("SELECT credito FROM Utente WHERE id = $id");
2
3 $riga = mysql_fetch_assoc($query);
4 if($riga['credito'] >= $_POST['somma']) {
5   ⟨esecuzione dell'operazione richiesta⟩
6   $nuovoCredito = $riga['credito'] − $_POST['somma'];
7   mysql_query("UPDATE Utente SET credito = $nuovoCredito WHERE id = $id");
8 }
```

Figura 3.1: Un esempio di un frammento di codice PHP vulnerabile.

che si verifichino dei comportamenti inaspettati. In questo lavoro viene mostrato che, a causa della semplificazione introdotta dai framework, i programmatori tendono ad ignorare totalmente i problemi di concorrenza nella realizzazione delle loro applicazioni web e che un utente maligno può indurre in un'applicazione un comportamento differente da quello originariamente inteso dal programmatore. In particolare questo lavoro si concentra sul rilevamento e sfruttamento di race condition che possono verificarsi a seguito dell'esecuzione di istanze multiple dello *stesso* script di un'applicazione web.

3.2 Analisi del problema

Per capire come il problema delle race condition si verifichi in ambiente web, si può analizzare il frammento di codice PHP riportato in Figura 3.1. In questo esempio, l'intento del programmatore è quello di implementare una procedura di trasferimento di fondi: l'utente prova a prelevare una `somma` di denaro; il sistema controlla se l'utente ha a disposizione una quantità sufficiente di soldi nel suo conto (linee 1–4) e, se disponibile, autorizza l'esecuzione dell'operazione richiesta (linea 5). Per finire, il sistema aggiorna l'account dell'utente prelevando la somma che era stata richiesta (linee 6–7).

Questa procedura sarebbe corretta solo se nessun altro thread o processo avesse accesso in lettura o scrittura alla stessa riga della tabella `Utente` finché la query alla linea 7 sia stata completamente eseguita. Si può facilmente verificare che un'esecuzione concorrente di istanze multiple dello stesso script può portare ad una violazione

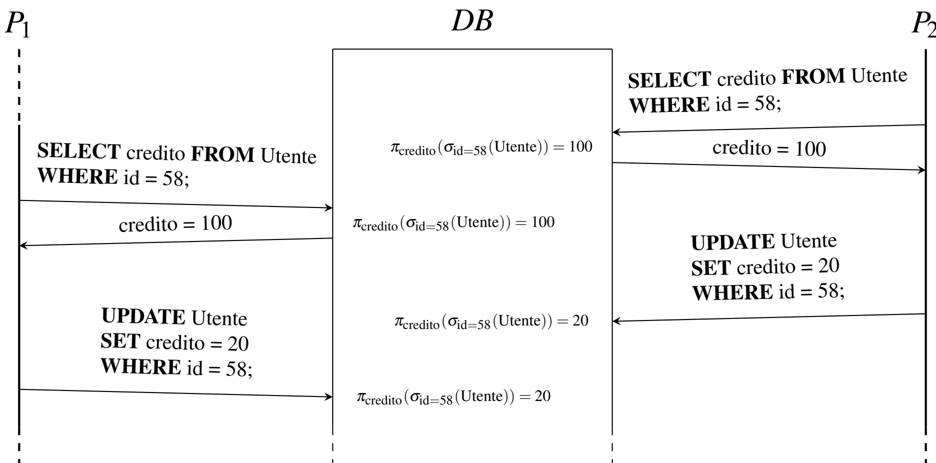


Figura 3.2: Un diagramma di sequenza di due thread o processi concorrenti che eseguono il frammento di script riportato in Figura 3.1.

d'integrità. Supponiamo infatti che un'istanza di questo script P arrivi ad eseguire l'istruzione alla linea 3, leggendo quindi dal database la riga t della tabella `Utente`: la procedura è soggetta ad una race se un'altra istanza riesce ad ottenere un accesso in lettura o scrittura alla riga t prima che P abbia completato l'esecuzione della query alla linea 7. L'esecuzione parallela di istanze multiple dello stesso frammento di codice produce una violazione della precondizione alla linea 4.

A seguito dell'esecuzione parallela di due istanze dello stesso pezzo di codice il sistema può entrare in un stato inconsistente, si consideri come esempio l'esecuzione riportata in Figura 3.2. Si assuma che un utente (identificato dall'id con valore 58) e con un credito di 100 euro esegua due istanze P_1 e P_2 del frammento di script di Figura 3.1. Si assuma inoltre che l'utente stia richiedendo un trasferimento da 80 euro. Se le due istanze sono schedulate dal sistema in modo tale che P_1 esegua la linea 1 mentre P_2 ha appena completato l'esecuzione della linea 6, nel database il credito dell'utente è ancora di 100 euro perché non è ancora stato eseguito nessun aggiornamento sul suo conto. A questo punto, se viene schedulato P_1 viene interrogato il database (linea 1) e concessa l'esecuzione del ramo *then* dell'istruzione *if* perché la condizione verificata alla linea 4 risulta ancora essere *vera*. Alla fine dell'esecuzione delle due istanze, anche se il credito iniziale dell'account era di 100 euro, l'utente è riuscito a completare due transazioni da 80 euro ciascuna, inoltre sul suo conto rimarranno ancora 20 euro

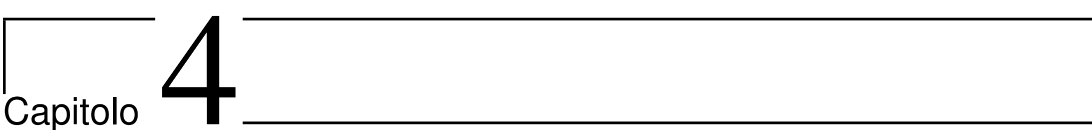
disponibili.

Questo è un classico problema di concorrenza di tipo *test & set* dovuto dall'assenza di meccanismi di sincronizzazione. Questi errori sono comuni in ambiente web perché *normalmente non si pensa all'applicazione web come multi-thread o multi-processo e si ignorano i problemi di concorrenza.* I framework non risolvono automaticamente tutti i problemi relativi alla concorrenza e in questi casi è compito del programmatore gestire in modo corretto le risorse condivise.

In questo lavoro viene mostrato un metodo per trovare in automatico errori di questo tipo e come un attaccante possa riuscire facilmente a sfruttarli. Il lavoro proposto si concentra sulle applicazioni PHP, ma i risultati sono indipendenti dal linguaggio ed il prototipo che è stato realizzato può essere facilmente esteso ad altre piattaforme (es., Perl, Python, Ruby). Inoltre, l'analisi è limitata alle race condition che possono verificarsi a causa di una incorretta gestione da parte dell'applicazione web del DBMS, ma simili problemi di concorrenza possono verificarsi anche con altre risorse condivise: il database è solo un esempio, anche se di fatto è la risorsa condivisa maggiormente utilizzata.

Le race condition possono portare a diversi problemi nell'esecuzione di un programma, ma non necessariamente portano a problemi di sicurezza. Negli esperimenti effettuati è stato possibile trovare un numero significativo di problemi di concorrenza e in generale la probabilità che almeno uno di questi difetti sia rilevante per la sicurezza è abbastanza alta.

Va inoltre sottolineato che la soluzione a questi problemi non può essere delegata completamente al DBMS o ai framework di programmazione. Tipicamente anche i più semplici DBMS forniscono delle primitive di sincronizzazione che permettono ai programmatori di gestire correttamente i problemi di concorrenza (es., istruzioni di locking, transazioni ACID) e che garantiscono effettivamente l'esecuzione atomica di ogni query o transazione passata al DBMS. Non è tuttavia possibile capire in automatico quando una sequenza di query dovrebbe essere eseguita in maniera atomica perché ciò dipende fortemente dalla logica dell'applicazione, per questo motivo è compito del programmatore usare le opportune primitive di sincronizzazione per evitare i problemi di concorrenza.



Rilevamento di race condition in applicazioni web

In questo capitolo viene analizzata l'architettura del metodo proposto per il rilevamento di race condition in applicazioni create con la piattaforma LAMP. La soluzione consiste nel realizzare un sistema che sia in grado automaticamente di individuare la presenza di una race condition all'interno di un'applicazione web e che possa aiutare il programmatore durante lo sviluppo. Sono discussi alcuni dettagli relativi all'implementazione e sono evidenziate le caratteristiche principali del prototipo sviluppato, insieme ad alcune limitazioni. Il capitolo si conclude con dei risultati sperimentali che sono stati elaborati ed alcune contromisure che si possono adottare per risolvere il problema delle race condition.

4.1 Il metodo di rilevamento

In questo lavoro ci si è concentrati sulle race condition che nascono da una incorretta gestione, da parte di un'applicazione web, del database utilizzato per la memorizzazione dei dati. Più in dettaglio, il metodo proposto si focalizza sull'individuazione di race che possono essere causate dall'esecuzione di istanze multiple dello stesso script.

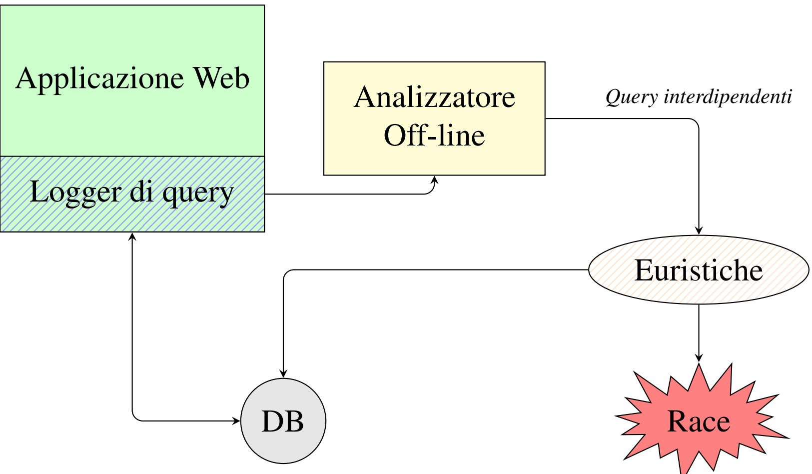


Figura 4.1: Architettura del framework per l'individuazione di race condition.

La strategia sviluppata è rappresentata in Figura 4.1 ed è composta dai seguenti componenti:

1. un *logger di query SQL*, che controlla l'esecuzione di ogni singola istruzione SQL e registra tutte le query che l'applicazione passa al DBMS;

2. un *analizzatore off-line*, che esamina i file prodotti dal logger di query SQL e individua le query che sono potenzialmente pericolose. Questo componente è suddiviso in due moduli: il primo cerca nei file di log le query interdipendenti che sono un indicatore della possibile presenza di una race condition; il secondo migliora i risultati ottenuti rimuovendo le query che sicuramente non sono soggette a questo problema.

4.2 Analisi dettagliata dei componenti

4.2.1 Logger di query SQL

Ci sono differenti metodi che possono essere utilizzati per registrare le query eseguite da un'applicazione web: è possibile intercettarle a livello di DBMS, in alternativa si può modificare il modulo utilizzato dall'interprete dell'applicazione per comunicare

con il DBMS; un terza possibilità consiste nell'intercettare le query SQL a livello applicativo, creando un *wrapper* sulle funzioni che utilizzano il database. Il prototipo realizzato utilizza quest'ultima strategia. Sebbene tutti e tre i metodi siano possibili in ambiente open-source, l'ultima strategia è l'unica possibile in ambiente closed-source dove tipicamente non è possibile modificare né il DBMS né il codice dell'interprete. La necessità di avere il codice sorgente dell'applicazione da analizzare non è un problema, il tool realizzato è infatti pensato per gli sviluppatori che vogliono eliminare i problemi di concorrenza dalle loro applicazioni. Questo strumento può comunque essere utilizzato su applicazioni open-source anche per fare vulnerability assessment. Per poter registrare le query è necessario che, durante l'esecuzione, ogni volta che l'applicazione richiama una funzione relativa al database, il modulo di logging intercetti la stringa contenente la query. Successivamente, la query viene memorizzata all'interno di un file di testo per poter poi essere utilizzata dall'analizzatore off-line.

4.2.2 Analizzatore off-line: approccio semplice

Dopo aver memorizzato le query eseguite dall'applicazione, i file di log creati sono esaminati da un apposito programma. L'idea è quella di individuare le query interdipendenti in modo da poter trovare delle race condition. Più precisamente, sia $q = \{s_1, s_2, \ldots, s_n\}$ una query, dove s_i indica un *oggetto* presente nello schema del database (attributo o tabella) utilizzato da q. Un *oggetto* viene considerato *usato* da una query quando il suo valore è letto; un *attributo* è *definito* da una query quando il suo valore è modificato dall'esecuzione di un comando; una *tabella* invece è considerata *definita* da una query se la sua esecuzione modifica parzialmente o totalmente le tuple presenti in quella tabella. Ad esempio, un'istruzione di **DELETE** *definisce* la tabella che appare dopo la clausola **FROM**, mentre *usa* qualsiasi oggetto che appare nella clausola **WHERE**.

Data una query q, viene definito *use(q)* e *def(q)* l'insieme degli oggetti che sono rispettivamente usati e definiti da q. Possiamo quindi formalizzare la nozione di *interdipendenza* con la seguente definizione:

Input: $Q = \{q_i, i = 1, 2, \ldots, n\}$, una lista di query SQL.
Output: $R = \{(p,q), p \in Q \wedge q \in Q\}$, una lista di coppie di query SQL che potrebbero essere soggette ad una race condition.

```
R = ∅
for j = 2,3,...,n do
    q = Q[j]
    D = def(q)
    for i = j−1, j−2,...,1 do
        p = Q[i]
        U = use(p)
        if U ∩ D ≠ ∅ then
            R = R ∪ {(p,q)}
```

Figura 4.2: Pseudo-codice per una versione semplificata dell'algoritmo di rilevamento di query interdipendenti.

Definizione Siano (p,q) una coppia di query SQL. Allora, (p,q) sono dette *interdipendenti* se: $use(p) \cap def(q) \neq \emptyset$.

Due query interdipendenti possono portare ad una race condition, per questo motivo la strategia adottata consiste nel trovare un insieme di coppie di query che sono interdipendenti.

Definizione Sia $Q = \{q_1, q_2, \ldots, q_n\}$ un insieme di query SQL. Si definisca una relazione di ordine totale $<$ sui suoi elementi, tale che $\forall q_i, q_j \in Q, q_i < q_j$ se e solo se $i < j$, in altri termini, q_i appare *prima* di q_j nel file di log.

Definizione L'insieme R delle *coppie di query interdipendenti* è definito come:

$$R = \{(q_i, q_j) \in Q \times Q \mid (q_i < q_j) \wedge (use(q_i) \cap def(q_j) \neq \emptyset)\}.$$

L'algoritmo riportato in Figura 4.2 formalizza queste nozioni, riceve come input una lista di istruzioni SQL e restituisce un insieme di coppie di query che sono interdipendenti tra di loro. Per ogni coppia individuata può esserci una race condition.

Come esempio, si consideri il frammento di log riportato in Figura 4.3. Si supponga che, per prevenire un attacco di brute force, un'applicazione web controlli se l'uten-

```
SELECT login_falliti
FROM Utente
WHERE id_utente = 123;

UPDATE Utente
SET login_falliti = login_falliti + 1
WHERE id_utente = 123;
```

Figura 4.3: Un esempio di un frammento di log con due query SQL interdipendenti.

te che sta cercando di autenticarsi ha già eseguito troppi tentativi di login senza successo. Con la prima query (l'istruzione **SELECT**), l'applicazione legge dalla tabella Utente il valore dell'attributo login_falliti relativo all'utente identificato da $id_utente = 123$. Successivamente il programma controlla se il valore letto è inferiore di una costante $\mathbb{K}$ predefinita; si noti che, anche se questo comportamento può essere dedotto, non è visibile dal file di log. Infine, se il valore dell'attributo login_falliti è inferiore del valore massimo di tentativi consentiti e l'autenticazione fallisce, l'applicazione esegue la seconda query (l'istruzione **UPDATE**) per aggiornare il numero di fallimenti. Analizzando queste query possiamo determinare che la prima *usa* l'attributo login_falliti e la tabella Utente mentre la seconda *definisce* l'attributo login_falliti. È quindi presente nel codice un classico problema time-of-check-to-time-of-use (TOCTTOU) [24, 3]: nell'intervallo di tempo tra il completamento della prima query e l'esecuzione della seconda, un'altra istanza dello stesso script potrebbe raggiungere il punto del programma nel quale viene eseguita la prima query **SELECT**, ottenendo così una valore obsoleto per l'attributo login_falliti, che non è ancora stato aggiornato dalla seconda query di **UPDATE**. In questa situazione un utente maligno può oltrepassare i controlli dell'applicazione presenti per limitare il numero di tentativi di login inviando n richieste parallele con $n > \mathbb{K}$. Quando queste richieste raggiungono il server web vengono create n differenti istanze dello stesso script, se l'attaccante è fortunato tutte queste istanze eseguono l'istruzione **SELECT** prima che qualsiasi richiesta raggiunga l'istruzione di **UPDATE** . In questo modo l'attaccante è in grado di eseguire n tentativi di login e non solamente $\mathbb{K}$.

4.2.3 Analizzatore off-line: riduzione dei falsi positivi

Alcune delle coppie di query individuate possono rappresentare dei falsi positivi. Per questo motivo, è stato sviluppato un modulo in grado di rimuovere le coppie di query interdipendenti che non sono soggette a race condition: il modulo è composto dalle euristiche riportate di seguito.

4.2.3.1 Clausole WHERE

Con l'utilizzo del metodo base, si ha una fonte rilevante di falsi positivi dovuta a query che sono interdipendenti ma che in realtà lavorano su insiemi di dati disgiunti a causa delle condizioni presenti nelle clausole **WHERE**.

Come esempio, si considerino le query SQL riportate di seguito:

```
SELECT id_utente
FROM Sessione
WHERE scadenza >= 1250678346;

DELETE FROM Sessione
WHERE scadenza < 1250678346;
```

In questo caso l'applicazione estrae dalla tabella Sessione gli *id* degli utenti la cui sessione è ancora valida; successivamente l'applicazione rimuove dal database le sessioni scadute. Apparentemente una race condition è presente tra queste due query perché la prima usa la tabella Sessione (e l'attributo id_utente) mentre la seconda la definisce. Si tratta però di un falso positivo perché l'intersezione delle tuple selezionate dalle clausole **WHERE** delle due istruzioni è *sempre* un insieme vuoto.

Per risolvere questo problema è necessario avere a disposizione un meccanismo che possa determinare quando le clausole **WHERE** identificano due insiemi disgiunti di tuple. In questa situazione non può essere presente una race condition, anche se le query sono interdipendenti. Si assuma che q_1, q_2 siano due query che condividono la stessa clausola **FROM** f, ma abbiano differenti clausole **WHERE** w_1, w_2

Un possibile approccio per l'identificazione di queste query è quello di interrogare dinamicamente la base di dati. Tutte le volte che è necessario capire se l'intersezio-

ne tra gli insiemi di tuple identificate dalle condizioni w_1 e w_2 sia vuota, è possibile costruire questa istruzione SQL:

$$\textbf{SELECT} * \textbf{FROM } f \textbf{ WHERE } w_1 \textbf{ AND } w_2$$

Se l'insieme di tuple restituite da questa istruzione *non* è vuoto allora possiamo affermare che c'è una race condition tra le query q_1 e q_2. Se invece l'insieme restituito è vuoto, si può solo assumere che non ci sia una race condition sull'istanza corrente del database ovvero tra i dati che sono contenuti al momento dell'esecuzione dell'istruzione; non si ha tuttavia la certezza che nessuna race sia presente sulle query individuate.

Un approccio alternativo, consiste nell'utilizzare un constraint solver per determinare se l'intersezione degli insiemi di tuple identificati dalle condizioni w_1 e w_2 sia effettivamente disgiunto. Questo metodo ha l'ovvio vantaggio di ragionare su *qualsiasi* possibile istanza del database, e non solo su quella corrente, superando così il principale svantaggio del precedente approccio. Si introduce però un overhead significativo a causa dell'utilizzo di un constraint solver esterno. Inoltre, è importante notare che il constraint solver non è in grado di gestire alcune istruzioni SQL, fra cui l'espressione **LIKE** o le query annidate. In queste situazioni il constraint solver dovrebbe comportarsi in maniera conservativa riportando che, sulle query analizzate, non è possibile escludere la presenza di una race condition. Tuttavia in molte situazioni reali, nelle quali queste istruzioni complesse non sono presenti, questa tecnica è utilizzabile ed efficace.

Si osservi che l'approccio basato su constraint solver e quello sull'interrogazione del database sono complementari e non alternativi: questi due metodi possono essere impiegati insieme per combinare l'efficienza dell'interrogazione diretta con la completezza del constraint solver. Tutte le volte che una possibile race è individuata, il DBMS può essere interrogato dinamicamente per verificare se, nell'istanza corrente, gli insiemi delle tuple selezionate dalle due clausole **WHERE** sono disgiunti oppure no. Se non vengono trovate tuple in comune, si può passare al metodo meno efficiente basato sul constraint solver per ottenere una risposta certa, o quanto meno conservativa.

4.2.3.2 Associazione degli attributi alle tabelle

Osservando una singola query SQL, non è sempre possibile dedurre a quale tabella appartengano gli attributi utilizzati: questa è un'altra importante fonte di falsi positivi. Si consideri, ad esempio, la query:

$$\textbf{SELECT } a_1, a_2 \textbf{ FROM } T_1, T_2$$

L'attributo a_1 potrebbe appartenere sia alla tabella T_1 che alla tabella T_2. L'unica cosa che è possibile fare in questi casi è assumere conservativamente che entrambi gli attributi possano appartenere a qualsiasi tabella presente nella query analizzata. Ovviamente questo può introdurre dei falsi positivi. In questi casi, per superare questa limitazione, l'analizzatore si occupa di interrogare il DBMS per determinare a quale tabella appartenga realmente ogni singolo attributo. Nell'interrogazione **SELECT** dell'esempio precedente, l'analizzatore recupera dal DBMS la struttura delle due tabelle T_1 e T_2 riuscendo così ad associare gli attributi a_1 e a_2 alle relative tabelle.

4.2.3.3 Annotazioni

L'algoritmo presentato precedentemente per il rilevamento di race condition non tiene conto di nessun tentativo esplicito di sincronizzazione. Verrà discusso più avanti, nella Sezione 4.4 questa particolare scelta di progettazione. La principale conseguenza di questa limitazione è che il rilevatore continuerà a riportare la presenza di una race condition anche quando sono stati adottati gli accorgimenti necessari per impedire che questa si verifichi. Ovviamente questo comportamento limiterebbe pesantemente l'utilizzo di questo strumento in un ambiente reale di sviluppo.

L'identificazione accurata di un possibile tentativo di sincronizzazione, senza sapere a priori quale insieme di primitive vengano utilizzate, è un compito complesso. Questo risulta del tutto impossibile con il meccanismo proposto in quanto le primitive di sincronizzazione possono essere richiamate anche a livello applicativo e, osservando solo le query utilizzate dall'applicazione, è impossibile sapere se nel codice siano presenti delle istruzioni di sincronizzazione. Per questo motivo è stato previsto un meccanismo di annotazioni che permette al programmatore di specificare esattamente

```
#! TAG get_all_ids
SELECT id_utente
FROM Utente

#! SAFE get_all_ids
DELETE FROM Utente
WHERE id_utente = 10
```

Figura 4.4: Un esempio di annotazione e rimozione dal report di race individuate.

che una race condition tra una coppia di query è stata risolta e che l'analizzatore non deve più segnalare nessun errore su quella coppia. Le annotazioni che sono state gestite dall'analizzatore creato iniziano con il prefisso `"#!"`. Il carattere `'#'` indica che la linea corrente contiene un commento fino a fine riga[1], in questo modo le annotazioni sono totalmente ignorate dal DBMS. Il carattere `'!'` permette all'analizzatore di distinguere le annotazioni dai normali commenti. Sono stati introdotti due differenti tipi di annotazioni:

`TAG` <***nome***> un'annotazione di questo tipo permette al programmatore di identificare univocamente ogni query assegnandogli un nome univoco;

`SAFE` <***nome***> questa annotazione specifica che ogni eventuale race tra la successiva query che verrà analizzata e quella identificata dall'etichetta <*nome*> non deve essere riportata.

Come esempio si considerino le query riportate in Figura 4.4, il programmatore ha assegnato alla prima interrogazione **SELECT** il nome get_all_ids usando l'annotazione di tipo *TAG*. Successivamente, è stata utilizzata l'annotazione di tipo *SAFE* per far si che l'analizzatore non riporti nessun errore per la race presente nella coppia di query. Grazie alle annotazioni un programmatore può facilmente provare la sua applicazione web integrando o meno il modulo di analisi, risolvere i problemi trovati, annotare le query e concentrarsi sulle nuove race evitando che quelle risolte siano riportate nuovamente.

[1] Questo carattere è utilizzato per la gestione dei commenti in molti DBMS, ad esempio da MySQL. Se un DBMS utilizza un altro marcatore è necessario utilizzare il suo carattere specifico per i commenti.

4.3 Implementazione

Il metodo di rilevamento descritto è stato implementato in un prototipo che supporta applicazioni scritte in PHP che utilizzano MySQL come DBMS. L'implementazione del logger delle query è effettuata con alcuni wrapper PHP realizzati in modo da sostituire le funzioni **mysql_query**() e **mysqli_query**().

La prima operazione da fare, per poter analizzare un'applicazione web, è la sostituzione di ogni chiamata alle funzioni per l'esecuzione di query con i relativi wrapper. Molte applicazioni web, soprattutto le più complesse, utilizzano una classe o una libreria per interfacciarsi con il DBMS, in questo modo riescono ad astrarre l'applicazione dal tipo di DBMS utilizzato. In questi casi, per poter integrare il sistema sviluppato, è sufficiente modificare un numero molto limitato di istruzioni. Questa operazione può essere del tutto automatizzata tramite un semplice script. Il wrapper realizzato memorizza, in un file di testo, tutte le query che sono sottomesse al DBMS insieme ad alcune meta-informazioni, come ad esempio il nome dello script analizzato e un dump dello stack-trace prima di effettuare la query. Quando una race è individuata, tutte le meta-informazioni raccolte aiutano il programmatore a localizzare facilmente in quale parte del codice si trova il problema riportato e che flusso di esecuzione è stato eseguito.

Dopo aver memorizzato tutte le query che sono state eseguite dall'applicazione, i file di log sono utilizzati dall'analizzatore per trovare le possibili race condition. L'analizzatore sviluppato è composto da circa 2.000 righe di codice Python ed implementa la strategia di detection discussa nella Sezione 4.2. L'unica parte che non è presente nel prototipo realizzato è quella relativa al constraint solver necessario per determinare se le condizioni presenti nelle clausole **WHERE** di una coppia di query identificano sempre due insiemi disgiunti di tuple. Per il parsing delle istruzioni SQL l'analizzatore fa affidamento sul modulo PERL `DBIx::MyParsePP`.

4.4 Criticità dell'architettura

L'algoritmo proposto per la rilevazione di race condition presenta le seguenti limitazioni:

- l'approccio utilizzato è *completamente dinamico*, per questo motivo le analisi possono essere svolte soltanto su di uno *specifico* percorso di esecuzione, quello osservato durante la fase di logging.
- Non viene utilizzata nessuna informazione riguardo la semantica dell'applicazione, le uniche informazioni trattate sono le query intercettate. Per esempio, non viene considerato come i dati recuperati dal database siano poi utilizzati.
- L'algoritmo proposto non considera alcun tentativo esplicito di sincronizzazione che l'applicazione può aver adottato per risolvere i problemi di concorrenza.

La prima limitazione può essere superata con l'utilizzo di tecniche di analisi statica o ibrida sul sorgente del programma: con l'utilizzo di queste tecniche è possibile analizzare più percorsi di esecuzione invece che osservarne solo uno. L'uso di metodi di analisi statica su linguaggi di scripting orientati agli oggetti e dinamici, come PHP, è tutt'altro che semplice e richiede la risoluzione di problemi molto complessi [20]. Per esempio, l'analisi di applicazioni PHP richiede l'esecuzione di analisi di "points-to" e di aliasing, che, in generale, sono dei problemi indecidibili [19]. L'impiego di tecniche di analisi statica permetterebbe inoltre di ottenere più informazioni sulla semantica dell'applicazione superando così anche la seconda limitazione riportata.

L'ultima limitazione elencata riguarda la mancanza del supporto delle primitive di sincronizzazione. Più che una limitazione questa è stata una scelta di progettazione e viene discussa di seguito. Prima di tutto, a livello applicativo, PHP non fornisce alcuna primitiva esplicita di sincronizzazione portabile su diversi sistemi operativi che possa essere utile allo scopo. Per esempio, PHP supporta la funzione **flock**() la quale implementa un meccanismo di locking che può essere utilizzato per la sincronizzazione e la mutua esclusione, però, come descritto nel manuale [29], su alcuni sistemi operativi **flock**() è implementata a livello di processo. Se il server web utilizza più thread all'interno di un singolo processo per eseguire i singoli script, la funzione **flock**() diventa

del tutto inefficace. Inoltre, **flock**() blocca il chiamante fino a quando il lock sul file non viene rilasciato, a meno che sia specificato il flag `LOCK_NB`: tuttavia tale funzionalità non risulta essere supportata su sistemi Windows. PHP mette a disposizione dei wrapper per le funzioni System V IPC, ma questa libreria non è abilitata di default e non è disponibile sulle piattaforme Windows.

In secondo luogo, a livello di database, la disponibilità di primitive di sincronizzazione è fortemente dipendente dal DBMS e spesso quelle presenti lavorano ad un livello troppo grossolano. Ad esempio, MySQL, probabilmente il database open-source più utilizzato, supporta le istruzioni **LOCK TABLES** e **UNLOCK TABLES** che forniscono un meccanismo di locking a livello di tabella. Purtroppo la granularità con cui lavorano è troppo grossolana per poter essere utilizzate in applicazioni web con alti carichi. MySQL supporta inoltre le transazioni ACID con un meccanismo di locking a livello di tupla, ma questa funzionalità non è disponibile quando viene utilizzato come motore di archiviazione `MyISAM` che è quello di default [2] e maggiormente utilizzato. Un meccanismo più efficiente è dato dall'utilizzo della funzione **GET_LOCK**() [28], che può essere utilizzata per la creazione di lock con nome. Se un nome è stato già acquisito da un client, **GET_LOCK**() blocca qualsiasi richiesta ricevuta da un altro client con lo stesso nome. Questo meccanismo permette ai client di accordarsi su di un certo valore ed eseguire un'operazione di locking cooperativa, il lock può essere rilasciato chiamando la funzione **RELEASE_LOCK**(). Purtroppo, anche questa soluzione non è semplice da gestire poiché con questa funzione è possibile ottenere un solo lock per connessione. Per ottenere più lock durante l'esecuzione di uno script, è necessario creare una nuova connessione ogni volta, prima di richiamare la funzione **GET_LOCK**(), aumentando così l'overhead dell'applicazione.

Nonostante alcune soluzioni riportate possano risolvere concretamente i problemi di concorrenza, la responsabilità del loro corretto utilizzo è delegata totalmente ai programmatori che dovrebbero conoscere il problema e utilizzare delle esplicite policy di sincronizzazione. Bisogna considerare che, in alcune situazioni, la penalizzazione dovuta all'utilizzo di questi meccanismi di sincronizzazione non può essere accettabile perché ridurrebbe drasticamente le performance dell'applicazione. Negli esperimen-

[2] Dalla versione 3.23 fino alla versione 5.5.4

ti condotti per questo lavoro si è osservato che le primitive di sincronizzazione sono raramente utilizzate e l'assenza della loro gestione, nel prototipo sviluppato, non ha prodotto alcun falso positivo.

4.5 Valutazione del prototipo

Per verificare l'efficacia del prototipo realizzato per il rilevamento di vulnerabilità, è stata eseguita l'analisi di alcune applicazioni open-source reali largamente utilizzate in ambienti di produzione (es., phpBB3 [31] e Drupal [38]). Ovviamente, il problema principale nella valutazione del prototipo, come per qualsiasi altro strumento di analisi dinamica, riguarda la raccolta di tracce di esecuzione significative. La capacità di individuare delle race condition sconosciute dipende fortemente dalla copertura dei percorsi di esecuzione durante la fase di logging. Negli esperimenti effettuati si è cercato di analizzare il comportamento di un'applicazione web simulando l'utilizzo da parte di un normale utente. Per esempio, con un forum si è provato ad effettuare il login fornendo sia credenziali corrette che sbagliate, si è provato ad aggiungere nuovi utenti, leggere alcune discussioni, crearne delle nuove, creare sondaggi, spedire messaggi privati ad altri utenti e così via. Tutte le applicazioni testate sono risultate soggette a problemi di concorrenza. Anche se molti di questi difetti non possono portare alla compromissione della logica dell'applicazione, alcuni di questi permettono ad un utente maligno di violare alcune proprietà di sicurezza.

Il prototipo realizzato è stato eseguito su una macchina Linux con un processore

Applicazione	Categoria	Query	Tempo	FP	VP
Drupal 7.0	CMS	13416	848.61 s	0	63 (3)
phpBB 3.0.8	forum	3136	93.11 s	0	58 (4)
WordPress 3.1	blog/CMS	3729	98.35 s	0	82 (3)
Magento 1.5.0.1	carrello virtuale	9453	1341.44 s	0	61 (2)

Tabella 4.1: Valutazione del prototipo realizzato. FP: Falsi Positivi; VP: Veri Positivi (le race rilevanti per la sicurezza sono riportate nelle parentesi). Questi risultati sono stati ottenuti senza l'utilizzo delle annotazioni supportate dal sistema.

dual-core Pentium a 2.0 GHz e 3GB di RAM. Nella Tabella 4.1 sono riportati i risultati ottenuti durante gli esperimenti effettuati. Il tempo impiegato per analizzare i file di log contenenti le query delle applicazioni è abbastanza lungo, ma più del 95% del tempo di esecuzione è stato utilizzato per il parsing delle istruzioni SQL. Questo overhead è dovuto principalmente alla comunicazione inter-processo tra l'analizzatore Python e il modulo Perl utilizzato per il parsing. L'overhead per il logging delle query SQL è trascurabile e non è stato riportato nella Tabella 4.1.

Come già discusso precedentemente, non tutte le race condition trovate sono rilevanti per la sicurezza, va però sottolineato che il numero di race trovate relative alla sicurezza insieme all'assenza di falsi positivi, dimostrano la validità del metodo proposto e del prototipo realizzato. Le vulnerabilità riguardanti la sicurezza che sono state individuate possono essere classificate come segue.

Risorse limitate: il più classico dei problemi relativi alle race condition riguarda la gestione di risorse limitate. In ogni caso in cui l'applicazione controlla la presenza di una risorsa, esegue l'operazione richiesta e diminuisce la risorsa stessa, può verificarsi una race condition. A questa categoria appartiene la procedura di trasferimento credito ampiamente descritta nella Sezione 3.2. Altri esempi di applicazioni che soffrono di questo problema sono quelle che offrono dei "buoni sconto" o che utilizzano dei token validi una sola volta per offrire qualche vantaggio all'utente.

Utenti multipli: ad esclusione di qualche caso particolare, tutte le applicazioni analizzate sono vulnerabili ad una race condition sul controllo dell'unicità dello username: un utente maligno può registrare account multipli con lo stesso username, oltrepassando i controlli dell'applicazione. L'impatto sulla sicurezza di questa vulnerabilità è fortemente dipendente dall'applicazione.

Brute forcing: alcune applicazioni (es., phpBB3), per prevenire eventuali attacchi di brute force, controllano se, un utente che sta cercando di autenticarsi, abbia già effettuato troppi tentativi incorretti. La procedura utilizzata per svolgere questa verifica contiene una race condition che può permettere ad un utente maligno di

oltrepassare i controlli presenti nell'applicazione. A seconda della logica dell'applicazione, questa vulnerabilità può consentire ad un attaccante di effettuare solo un numero limitato di tentativi aggiuntivi (es., quando il codice dell'applicazione controlla che `provaCorrente` $\leq$ `NUMERO_MASSIMO_PROVE`), o di aggirare completamente il controllo applicativo (es., nel caso in cui l'applicazione utilizzi `provaCorrente` == `NUMERO_MASSIMO_PROVE`).

Voti multipli sui sondaggi: i forum web e i CMS spesso offrono la possibilità di creare dei sondaggi. Queste applicazioni cercano di verificare che ogni utente possa esprimere solo una preferenza per ogni sondaggio proposto. Purtroppo ogni programma analizzato è vulnerabile ad una race condition che permette ad un utente di votare più volte inviando delle richieste di voto parallele.

Topic flooding: phpBB3 e WordPress implementano una funzionalità anti-flooding obbligando un utente, che ha appena inserito un messaggio, ad aspettare qualche secondo prima che possa inserirne un altro. Anche questa verifica può essere facilmente aggirata da un attaccante per la mancanza di controlli di sincronizzazione.

Nelle applicazioni web analizzate sono stati rilevati pochissimi tentativi di sincronizzazione. Questo conferma che i programmatori non si preoccupano o non sono a conoscenza del potenziale impatto dei problemi di concorrenza sulle loro applicazioni.

4.6 Contromisure

In questa sezione sono presentate delle possibili contromisure che possono essere adottate dai programmatori per mitigare il problema delle race condition.

Come è stato mostrato, un approccio per la risoluzione di questo problema consiste nell'utilizzo, quando disponibili, delle primitive di sincronizzazione messe a disposizione dal linguaggio di programmazione o dal DBMS. In questi casi bisogna sempre tenere in considerazione tutte le limitazioni presenti per poter garantire la portabilità del proprio codice e salvaguardare al massimo le risorse disponibili.

```
1  mysql_query("LOCK TABLES Utente WRITE");
2  $query = mysql_query("SELECT credito FROM Utente WHERE id = $id");
3
4  $riga = mysql_fetch_assoc($query);
5  if($riga['credito'] >= $_POST['somma']) {
6    $creditoAttuale = $riga['credito'] − $_POST['somma'];
7    mysql_query("UPDATE Utente SET credito = $creditoAttuale WHERE id=$id");
8    mysql_query("UNLOCK TABLES");
9
10   ⟨esecuzione dell'operazione richiesta⟩
11   if ($transazioneFallita){
12     $somma = $_POST['somma'];
13     mysql_query("UPDATE Utente SET credito = credito + $somma WHERE id=$id");
14   }
15 } else {
16   mysql_query("UNLOCK TABLES");
17 }
```

Figura 4.5: Semplice algoritmo di two-phase commit.

Nella Sezione 4.4 sono state evidenziate le limitazioni dell'ambiente PHP/MySQL. Ovviamente framework differenti possono offrire dei meccanismi di locking più granulari (es., locking a livello di riga), ma questo tipicamente implica una minore efficienza ed un consumo maggiore di risorse.

In alcuni casi è possibile risolvere il problema con qualche piccolo accorgimento. Per esempio, una soluzione basata sul locking dell'intera tabella è sicuramente una soluzione troppo inefficiente da applicare all'esempio di Figura 3.1: in questo caso l'operazione potrebbe richiedere del tempo per essere eseguita e l'applicazione non può essere costretta a bloccare ogni altra richiesta per tutto il tempo necessario al suo completamento. In questa situazione esiste una tecnica che, con lo stesso meccanismo di locking, può portare ad una parziale soluzione del problema. È possibile infatti spostare l'istruzione **UPDATE** prima dell'esecuzione della transazione che richiede tempo, introdurre l'istruzione di locking della tabella prima della **SELECT** iniziale e infine aggiungere l'istruzione per sbloccare la tabella sia dopo l'istruzione di **UPDATE** sia nel ramo *else* della condizione. Il codice risultante è riportato in Figura 4.5. Questa soluzione è un semplice algoritmo di two-phase commit che richiede una procedura

addizionale per la gestione degli errori: il credito è correttamente prelevato dal conto se la transazione va a buon fine, ma deve essere restituito se la transazione fallisce. Bisogna sottolineare che questa soluzione non può sempre essere utilizzata perché non equivale all'implementazione di una transazione *ACID*, non c'è infatti nessuna garanzia sull'atomicità dell'insieme di istruzioni riguardanti la transazione.

La gestione corretta e automatica dei problemi di concorrenza con l'utilizzo di appositi meccanismi di locking, senza influenzare le prestazioni dell'applicazione, è un compito complesso. Sarebbe infatti abbastanza semplice inserire in automatico delle istruzioni di locking attorno alle regioni critiche, ma risulterebbe molto complicato riuscire ad evitare deadlock ed un degrado sensibile delle prestazioni.

Race condition in applicazioni web reali

In questo capitolo viene approfondito l'impatto delle race su applicazioni web reali. A partire dall'analisi teorica sviluppata nella prima parte di questo lavoro, sono state individuate delle strategie per sfruttare questa vulnerabilità. Si riportano i risultati ottenuti attraverso alcuni esperimenti fatti su applicazioni esistenti, seguiti da una breve panoramica relativa alle caratteristiche della vulnerabilità. Viene infine illustrato un metodo particolare per poter realizzare un attacco.

5.1 Individuazione di race in software closed-source

Dopo aver analizzato nel Capitolo 3 un semplice esempio di race condition, in una ipotetica applicazione web, ci si potrebbe chiedere se questo difetto possa portare veramente a problemi di sicurezza su applicazioni web reali permettendo ad un attaccante di alterare la semantica di un'applicazione.

Le race condition in programmi tradizionali (es., C, Java) sono solitamente molto difficili da trovare e anche quando sono evidenziate da tool automatici è in genere necessaria una conoscenza approfondita dell'applicazione per poter capire le implicazioni di questo problema. Le applicazioni web, come già detto, sono tipicamente composte da un insieme di script indipendenti ognuno con una funzionalità ben precisa. Spesso sono utilizzati dei pattern di programmazione abbastanza semplici da capire o da prevedere anche in assenza del codice sorgente. Come esempio, si consideri una

qualsiasi applicazione che esegue qualche tipo di autenticazione a livello utente prevedendo una registrazione. Il processo di registrazione viene solitamente realizzato con l'utilizzo di un form nel quale l'utente inserisce i suoi dati. In seguito l'applicazione tipicamente interroga il database per controllare se un utente con la stessa username sia già presente e, nel caso non ci sia, lo inserisce. Questo è un pattern di programmazione abbastanza classico e può essere facilmente dedotto da un osservatore esterno solamente dallo studio dell'interfaccia fornita dall'applicazione. Per sfruttare questo meccanismo, su di un'applicazione vulnerabile, un utente maligno può fare due richieste parallele di registrazione con lo stesso username superando i controlli di unicità dell'applicazione. Ovviamente gli effetti generati da questa inconsistenza nel database sono fortemente dipendenti dall'applicazione. L'esempio dimostra che spesso i pattern di programmazione sono deducibili e un attaccante può individuarli anche senza avere accesso al codice sorgente. Di seguito viene mostrato un altro caso in cui è stato possibile dedurre il comportamento dell'applicazione senza il suo codice.

Per verificare l'impatto delle vulnerabilità derivate da race condition su applicazioni web reali sono stati eseguiti diversi esperimenti: in particolare sono stati presi in considerazione due sistemi commerciali senza essere in possesso del codice sorgente. Per la loro analisi si è utilizzata solo l'interfaccia web comune per tutti gli utenti. La prima applicazione è gestita da un Internet Service Provider italiano, mentre la seconda appartiene ad un provider internazionale di telecomunicazioni. Per ovvi motivi i nomi di queste due società non saranno rivelati. Entrambe le applicazioni consentono agli utenti di inviare SMS tramite un'interfaccia web, permettendo solo un numero limitato di messaggi per utente al giorno. Per entrambe le applicazioni l'ipotesi è stata la seguente: quando un utente autenticato prova a spedire un SMS, l'applicazione web controlla le informazioni relative all'account recuperandole dal database. È stato inoltre ipotizzato che venisse controllata la possibilità per l'utente di spedire il messaggio, che l'SMS venisse inoltrato verso qualche gateway per l'invio ed infine che le informazioni riguardanti l'utente fossero aggiornate per memorizzare l'esito dell'invio effettuato. Questo comportamento è molto simile a quello relativo alla procedura di trasferimento credito proposta in Figura 3.1. Fatte queste assunzioni si è provato a modificare la logica dell'applicazione: nel primo caso sono state spedite 11 richie-

ste parallele per l'invio degli SMS, il servizio ne avrebbe dovute accettare solo 10, ma sul cellulare, utilizzato per l'esperimento, sono arrivati tutti gli 11 SMS. Nel secondo caso sono state spedite 10 richieste in parallelo, l'applicazione avrebbe dovuto accettare solo la prima e scartare tutte le altre ma, anche in questo caso, sul cellulare sono arrivati tutti e 10 gli SMS. Questo semplice esperimento è una piccola dimostrazione di come ci siano in rete applicazioni web commerciali, e magari critiche, che sono vulnerabili a questo tipo di attacco. È inoltre interessante notare come sia stato possibile attaccare queste due applicazioni solamente effettuando delle congetture su come i programmatori possano aver realizzato l'applicazione, senza disporre del codice sorgente.

5.2 Caratteristiche della vulnerabilità

In questo lavoro si è mostrato che il problema delle race condition è presente anche in ambiente web: è possibile fare qualche paragone tra la pericolosità di questa vulnerabilità rispetto ad un'altra frequentemente presente in questo ambiente, la SQLI. Entrambe le vulnerabilità possono permettere ad un utente maligno di manipolare i dati contenuti nel database in una maniera imprevista. Un attacco di tipo SQLI ha come scopo quello di effettuare dinamicamente delle modifiche alle query che l'applicazione invia al DBMS. Al contrario, le race condition non modificano la struttura delle singole query SQL, ma modificano la sequenza con cui sono eseguite le query lecite presenti nell'applicazione. Le implicazioni di una vulnerabilità di tipo SQLI e di una race condition sono entrambe dipendenti dall'applicazione: non è sempre vero che una SQLI può portare alla compromissione di un'applicazione e non è sempre vero che una race condition sia rilevante dal punto di vista della sicurezza. Va sottolineato che gli attacchi di tipo SQLI sono largamente conosciuti e sono state proposte diverse soluzioni per la loro mitigazione, inoltre c'è una forte sensibilizzazione dei programmatori riguardo questo tema e i framework hanno adottato diverse tecniche per la loro risoluzione. Al contrario, le race condition e le loro implicazioni non sono mai state approfondite in ambito web nonostante il problema della concorrenza sia ben noto.

Lo sfruttamento delle vulnerabilità di tipo race condition richiedono spesso una profonda conoscenza della logica dell'applicazione e la realizzazione di un attacco risulta sicuramente più difficile rispetto alle altre categorie di vulnerabilità web, come quelle discusse nel Capitolo 2. Bisogna notare che la realizzazione di questo attacco non è totalmente deterministica dato che dipende dall'interleaving scelto dallo scheduler del sistema, cosa che non accade per i XSS o le SQLI. Malgrado ciò, durante gli esperimenti, è stato possibile alterare la semantica originale di *tutte* le applicazioni reali che sono state analizzate. Questo risultato è abbastanza sorprendente: l'individuazione di questi errori dimostra come il problema di fatto sia trascurato anche dai programmatori più esperti che mantengono codice open-source maturo e ampiamente utilizzato in ambienti di produzione.

5.3 Exploiting

Analizzando il problema delle race condition si potrebbe obiettare che lo sfruttamento di questa vulnerabilità su di un'applicazione web remota sia significativamente più difficile rispetto alla realizzazione dello stesso attacco su un'applicazione locale, a causa dei problemi di latenza introdotti dalla rete. In questa sezione viene proposta una tecnica particolare che, utilizzando alcuni dettagli del protocollo *HTTP*, riesce a ridurre l'impatto della rete sulla realizzazione dell'attacco.

Lo sfruttamento delle race condition nelle applicazioni tradizionali è tipicamente un compito abbastanza difficile dovuto alla natura non deterministica di questo tipo di difetto. Il successo di un tentativo di attacco spesso dipende da un particolare interleaving di istruzioni che sono scelte dallo scheduler, dal carico attuale del sistema e da un numero di altri fattori che solitamente non possono essere controllati dall'attaccante. Inoltre, l'attacco di una vulnerabilità remota è ulteriormente complicato dai seguenti fattori: la latenza introdotta dall'infrastruttura di rete, il tempo necessario al server per la creazione della connessione e il tempo impiegato per l'elaborazione dei dati inviati dal client. Affinché l'attacco abbia successo, l'attaccante deve assicurarsi che i messaggi spediti siano ricevuti e processati dall'applicazione vulnerabile qua-

si contemporaneamente, o in un altro ordine voluto. Alcuni accorgimenti possono semplificare la riuscita di un attacco.

Come esempio, si consideri una generica applicazione di consumo di risorse come quella descritta in Figura 3.1 e già ampiamente discussa nella Sezione 3.2. Questa applicazione (1) controlla che l'utente ha abbastanza risorse per effettuare l'operazione, se la condizione è verificata, (2) esegue l'operazione richiesta ed infine (3) toglie dall'account dell'utente una quantità k per le risorse che sono state utilizzate. Come già descritto precedentemente questo comportamento porta ad una vulnerabilità di tipo TOCTTOU. In questa situazione l'obiettivo dell'attaccante è quello di permettere al maggior numero di istanze n di raggiungere il punto (2) dell'applicazione web vulnerabile, per poter ottenere una quantità di risorse pari a $k \cdot n$.

In molte applicazioni di questo tipo, l'esecuzione del secondo punto potrebbe risultare più lenta rispetto al resto dell'applicazione a causa dell'utilizzo di qualche sistema a cui è necessario interfacciarsi (es., un web service remoto o un'applicazione legacy). Questo rallentamento allarga la finestra temporale per lo sfruttamento della vulnerabilità incrementando così la possibilità dell'attaccante di avere successo nel suo tentativo. Ad ogni modo, *non* è indispensabile la presenza di questo rallentamento affinché l'attacco abbia successo. Questo particolare può garantire il successo dell'attacco con un singolo tentativo invece di dover ricorrere ad un metodo più invasivo come un brute force. Infine, se la finestra dell'attacco è più ampia, è anche più semplice poter far accedere alla regione critica un maggior numero di richieste.

Per massimizzare la probabilità di successo di un tentativo di attacco su di un'applicazione vulnerabile ad una race condition relativa ad un consumo di risorse, un attaccante ha la necessità che le richieste inviate siano ricevute e processate dal server web praticamente contemporaneamente. Il problema principale dell'attaccante è quindi quello di ridurre al minimo la latenza introdotta dall'utilizzo della rete. Per risolvere questo problema è stata ideata la seguente strategia che sfrutta alcune particolarità delle specifiche del protocollo *HTTP* [15]: l'idea è quella di fare in modo che il server non processi le richieste dell'attacco fino a quando queste siano state *completamente* ricevute. Il meccanismo utilizzato per realizzare questa strategia è il seguente: quando un attaccante vuole spedire n richieste ad un server web remoto, inizia per prima cosa

Strategia	Ritardo Minimo	Ritardo Massimo	Ritardo Medio
Parallela	**8.537** ms	7039.140 ms	2538.249 ms
2-fasi	**6.353** ms	3023.917 ms	1217.697 ms
2-fasi ottimizzata	**1.468** ms	2932.940 ms	903.534 ms

Tabella 5.1: Valutazione del metodo proposto per la realizzazione dell'attacco. I valori sono stati calcolati effettuando 100 richieste *HTTP*.

a stabilire *n* differenti connessioni di rete tra il suo host e la macchina bersaglio. Successivamente, su ogni connessione creata, invia l'intera richiesta *HTTP* ad *eccezione* dell'ultimo carattere (es., l'ultimo LF per una richiesta di tipo *GET* o l'ultimo byte del `request body` per una richiesta di tipo *POST*). Queste *n* richieste *HTTP* parziali raggiungeranno la macchina target che allocherà tutte le risorse necessarie per gestire le connessioni e memorizzerà in un buffer temporaneo tutti i dati ricevuti. Infine, per ogni connessione creata, l'attaccante invierà l'ultimo carattere, completando così tutte le richieste. A questo punto il server potrà completare tutte le richieste parziali che erano state memorizzate temporaneamente e le passerà al framework o all'applicazione web. In questo modo, l'applicazione riceverà tutte le richieste contemporaneamente, riducendo la latenza introdotta dalla rete.

Nella Tabella 5.1 è stata comparata la strategia appena illustrata con altre due strategie che possono essere utilizzate per realizzare l'attacco. La tecnica "parallela" consiste nel creare un thread per ogni richiesta da effettuare e, successivamente, nell'inviare subito tutto il corpo della richiesta *HTTP* senza utilizzare particolari accorgimenti di sincronizzazione. Con la strategia "2-fasi", invece, in una prima fase viene creato un thread per ogni richiesta e viene effettuata la connessione con il server web, nella seconda fase la richiesta *HTTP* completa viene inviata al server web. Per finire, la strategia "2-fasi ottimizzata" corrisponde a quella descritta precedentemente. I risultati ottenuti sono stati calcolati considerando 100 richieste *HTTP*, indirizzate verso uno script remoto PHP che restituisce i secondi e i microsecondi correnti del server web, attraverso l'utilizzo della funzione **microtime** [30], in questo modo è possibile misurare il tempo in cui gli script vengono eseguiti.

In tutti i casi la connessione con il server web remoto è stata realizzata utilizzando direttamente l'indirizzo IP della macchina, affinché i risultati non venissero falsati

dall'overhead necessario per la risoluzione dei nomi di dominio.

La colonna `Ritardo Massimo` indica il massimo intervallo di tempo tra due richieste, mentre la colonna `Ritardo Medio` riporta l'intervallo di tempo in media tra due richieste. Il fattore più importante, per la riuscita di un attacco su una race condition, è il `Ritardo Minimo`: è necessario infatti che le due richieste arrivino contemporaneamente. I risultati in Tabella 5.1 mostrano che questo valore viene effettivamente diminuito con la strategia proposta, che si dimostra il miglior metodo per sfruttare questa vulnerabilità.

Un utente maligno può utilizzare la strategia descritta per la realizzazione di un attacco, ma bisogna notare che, anche durante una normale navigazione web, un utente qualsiasi potrebbe causare una race condition involontariamente. I moderni browser sono dei client multi-thread molto potenti ed ottimizzati per la navigazione; si supponga che l'utente si trovi davanti ad un normale form HTML e che abbia inserito dei dati: quando l'utente clicca sul bottone per inviarli, il browser fa subito partire una richiesta *HTTP*. In questa fase, se viene fatto un doppio click invece che un singolo click sul bottone di invio, oppure se l'utente clicca due volte perché la pagina risulta lenta a caricarsi, il browser genera due richieste *HTTP* identiche ed è possibile che si verifichi una race condition.

Lavori correlati

Le race condition sono probabilmente uno dei più antichi problemi legati al software e le loro implicazioni sono state discusse ampiamente nella letteratura [26, 18]. Nel corso degli anni è stato fatto molto lavoro di ricerca per la rilevazione di questo problema di concorrenza, sia per scopi di debugging che di sicurezza. In questo lavoro si è focalizzata l'attenzione sulle implicazioni che le race condition possono avere in ambito web. Nella seguente sezione vengono discusse delle soluzioni relative all'individuazione di race condition su programmi tradizionali, dal momento che non è stato pubblicato nessun lavoro che riguardi in modo specifico le applicazioni web.

6.1 Analisi statica

Sono state proposte diverse soluzioni che eseguono un'analisi del codice sorgente al momento della compilazione, per verificare la presenza di race condition in *qualsiasi* possibile esecuzione del programma [35, 14]. Altri approcci modificano i tipi del linguaggio di programmazione in modo tale che il linguaggio risultante sia garantito essere esente da race condition [5, 16]. Tipicamente, il maggiore svantaggio di questi strumenti è l'alto numero di falsi positivi: analizzare il codice sorgente di un'applicazione senza eseguirlo impone spesso di fare delle assunzioni conservative riguardo il possibile interleaving che si verifica a run-time.

6.2 Analisi dinamica

I metodi di analisi dinamica si basano sul monitoraggio di diverse esecuzioni del programma. Questi strumenti sono tipicamente facili da usare e più accurati rispetto a quelli di analisi statica, dato che possono osservare un'esecuzione reale dell'applicazione. Lo svantaggio di questi metodi è che non forniscono una risposta completa in quanto riescono a rilevare la presenza di un problema di sincronizzazione solo sui percorsi che sono stati eseguiti, ma non possono garantire l'assenza di race condition in tutti i possibili percorsi di esecuzione. Diversi metodi [13, 33] sono basati sul calcolo della relazione di Lamport *happens-before* [23], che produce un ordine parziale sulle istruzioni del programma. Altri approcci [8, 9] usano un'analisi basata su *lockset* [34], nei quali si assume che possa esserci una race condition perché il programmatore ha dimenticato di proteggere una variabile condivisa. In pratica ogni variabile condivisa è associata ad un *lockset* che contiene il lock acquisito durante l'accesso a questa variabile; se un lockset diventa vuoto allora può esserci una race condition. Sono stati proposti anche altri approcci [42, 32] che combinano i vantaggi di queste tecniche. Infine altri metodi dinamici [39] puntano a prevenire lo sfruttamento di race condition sulle operazioni del filesystem, tenendo traccia di ogni possibile interferenza tra le azioni svolte da processi differenti: se un'operazione sul filesystem risulta interferire con un'altra, il processo che l'ha generata è sospeso temporaneamente.

6.3 Model checking

Il model checking è una tecnica di verifica formale che è stata applicata anche alla rilevazione di problemi di concorrenza [7]. Un model checker riceve come input una versione semplificata del codice sorgente dell'applicazione ed esplora completamente tutti i possibili stati alla ricerca della violazione di alcune condizioni preimpostate. Per esempio, alcuni tool di model checking sono stati proposti per analizzare i problemi di concorrenza nei programmi Java [40]. Tuttavia, l'uso del model checking su sistemi software articolati è ancora molto complesso. Anche per sistemi più semplici è ri-

chiesto uno sforzo considerevole per la costruzione del modello semplificato che deve essere fornito allo strumento di analisi.

6.4 Classificazione della soluzione proposta

Il modello elaborato in questo lavoro può essere classificato come un metodo di rilevazione completamente dinamico. Va sottolineato che l'ambiente web ha delle particolarità che portano a problemi molto diversi rispetto a quelli discussi nei lavori citati. Mentre i programmatori web non si preoccupano delle implicazioni a cui la mancata gestione della concorrenza può portare, i programmatori di applicazioni tradizionali sono maggiormente abituati a trattare i problemi di sincronizzazione e ad utilizzare tutti i meccanismi necessari per evitarli. Tutti i lavori riportati in questa sezione sono focalizzati sull'analizzare il corretto utilizzo delle primitive di sincronizzazione, mentre il presente lavoro di tesi ha evidenziato il problema della concorrenza su istanze multiple di codice concepito come sequenziale che viene eseguito concorrentemente.

Conclusioni

In questo lavoro di tesi è stato affrontato il problema delle race condition in applicazioni web. Il problema della concorrenza è ben noto nel campo della sicurezza informatica, ma il suo impatto nelle applicazioni web non è stato sufficientemente esplorato. Si sono analizzati i problemi di concorrenza che si possono verificare nell'esecuzione di applicazioni che impiegano un DBMS senza l'utilizzo di primitive di sincronizzazione. La tesi mostra come, sfruttando l'esecuzione di istanze multiple di uno stesso script, un utente maligno possa alterare il comportamento di un'applicazione web rispetto a come originariamente inteso dal programmatore.

È stata formalizzata la relazione di dipendenza presente tra query che possono portare ad una race condition ed è stato proposto un metodo di rilevamento delle race condition basato sull'analisi dinamica. Per raffinare il processo di rilevazione sono poi state proposte delle euristiche che permettono di diminuire il numero di falsi positivi. È stato realizzato un prototipo, basato sulla strategia ideata, che ha permesso la rilevazione di diversi difetti, anche riguardanti la sicurezza, in software open-source maturi ed ampiamente utilizzati in ambienti di produzione.

Sono stati classificati i principali problemi di sicurezza che si possono verificare in un'applicazione web a causa di una race condition e sono state approfondite le possibili contromisure per la loro risoluzione. È stato infine analizzato il problema della concorrenza in applicazioni web reali mostrando che è possibile trovare race condition anche in applicazioni closed-source. Per poter realizzare un attacco su di un'appli-

cazione web remota è stata proposta una tecnica che, utilizzando alcuni dettagli del protocollo *HTTP*, è in grado di massimizzare la probabilità della riuscita di un attacco.

La strategia proposta in questo lavoro si è rivelata valida ed ha portato alla scoperta di diverse vulnerabilità sconosciute in software ampiamente utilizzati tuttavia vi sono margini di miglioramento. Una possibile evoluzione prevede l'estrazione di alcune meta informazioni dall'applicazione con tecniche di analisi statica per superare i problemi derivanti da un'analisi completamente dinamica riuscendo così a testare un maggior numero di percorsi di esecuzione. Inoltre è possibile trovare un maggior numero di race condition analizzando l'esecuzione contemporanea di differenti script della stessa applicazione che utilizzano i medesimi dati.

Bibliografia

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems.

[2] C. Alonso and R. Bordón. LDAP Injection & Blind LDAP Injection. *Blackhat Europe*, 2008.

[3] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 2(2):131–152, 1996.

[4] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2005.

[5] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 56–69, 2001.

[6] CERT. Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests, 2002.

[7] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques, July 23 1996.

[8] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *ACM SIGPLAN Notices*, 37(5):258–269, May 2002.

[10] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng;. Bandera: extracting finite-state models from java source code. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 439–448. IEEE, jun 2000.

[11] M. Cova, V. Felmetsger, and G. Vigna. Vulnerability Analysis of Web Applications. In L. Baresi and E. Dinitto, editors, *Testing and Analysis of Web Services*. Springer, 2007.

[12] D. Dean and A. J. Hu. Fixing races for fun and profit: How to use access(2). In *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[13] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–10, 1990.

[14] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[15] R. T. Fielding, J. Gettys, J. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[16] C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[17] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[18] D. Helmbold, C. McDowell, and S. C. C. R. L. University of California. *A taxonomy of race conditions*. Technical report (University of California, Santa Cruz. Computer Research Laboratory). Computer Research Laboratory [University of California, Santa Cruz, 1994.

[19] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001.

[20] N. Jovanovic. *Web Application Security*. PhD thesis, Technical University of Vienna, July 2007.

[21] A. Klein. Blind XPath Injection. *Whitepaper from Watchfire*, 2005.

[22] M. Kunze. Let there be light. LAMP: Freeware web publishing system with database support. *c't*, 12:230, 1998.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[24] W. S. McPhee. Operating System Integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.

[25] NCSA Software Development Group. *The Common Gateway Interface*, 1995.

[26] R. H. B. Netzer and B. P. Miller. What are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[27] Open Web Application Security Project. OWASP Top Ten Project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[28] Oracle. *MySQL Reference Manual*. http://dev.mysql.com/doc/refman/5.0/en/index.html.

[29] PHP Documentation Group. *PHP Manual*. http://www.php.net/manual/en/index.php.

[30] PHP Documentation Group. *PHP Manual - Microtime function*. http://www.php.net/microtime.

[31] phpBB Limited. phpBB - Free and Open Source Forum Software. http://www.phpbb.com/.

[32] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. *ACM SIGPLAN Notices*, 38(10):179–190, Oct. 2003.

[33] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions Computer Systems*, 17(2):133–152, 1999.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[35] N. Sterling. WARLOCK - A static data race analysis tool. In *Proceedings of the Usenix Winter 1993 Technical Conference*, pages 97–106, 1993.

[36] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[37] Symantec Inc. Symantec global internet security threat report: Volume XV. Technical report, Symantec Inc., apr 2010.

[38] The Drupal Association. Drupal - Open Source CMS. http://drupal.org/.

[39] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, Aug. 2003.

[40] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, Sept. 2000.

[41] Wikipedia. Samy (XSS). http://en.wikipedia.org/wiki/Samy_%28XSS%29.

[42] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. Technical report, Microsoft Research, Apr. 2005.