Università degli Studi di Genova

Corso di Studio in Ingegneria Informatica

Tesi di Laurea per il conseguimento del titolo di

Dottore Magistrale in Ingegneria Informatica

Definizione di scenari per il training di Web Security attraverso per-user sandboxing

Loris Piana

Dicembre 2017

Relatore: Prof. Alessio Merlo

Correlatore: Dott. Andrea Valenza

Indice

Introduzione

Oggigiorno il Web è diventato parte integrante della vita quotidiana di ognuno di noi. L'accesso alla rete si è reso ormai disponibile a tutti grazie alla diffusione, prima dei personal computer e, più recentemente, di smartphone e tablet. Di pari passo alla diffusione di questi strumenti tecnologici si sono moltiplicati anche i servizi offerti da Internet: se tra i primi servizi web vi erano soprattutto siti per gli acquisti online, oggi oltre a poter acquistare qualsiasi tipo di oggetto si possono pagare le bollette, effettuare operazioni bancarie, fare prenotazioni per alberghi, cinema, ristoranti e persino ordinare pizze, il tutto in pochi secondi e ovunque vi sia una connessione Internet.

Studi di settore mostrano come il numero di persone che utilizza questo tipo di applicazioni web sia in costante crescita e come non tutti gli utenti siano consapevoli dei rischi che l'utilizzo di esse può comportare. Infatti quando decidiamo di utilizzare una applicazione web accettiamo più o meno inconsciamente di affidare i nostri dati personali all'applicazione stessa che li raccoglie e li conserva nei propri database.

Questo ha portato negli ultimi anni le applicazioni web ad essere soggette ad attacchi informatici che hanno come obiettivo principale proprio i dati personali degli utenti che utilizzano tale applicazione. Ovviamente l'obiettivo più interessante per gli attaccanti sono quelle applicazioni che

prevedono l'utilizzo di carte di credito, ma in generale qualsiasi dato personale, anche la data di nascita di una persona, è comunque un buon bottino. Oltre ad attacchi mirati al furto di dati, spesso si verificano anche attacchi che hanno come obiettivo la modifica dei dati presenti nei database delle applicazioni e nei casi più estremi alla totale cancellazione. Questi attacchi sono spesso resi possibili da vulnerabilità presenti nella applicazione web che non sempre sono ben progettate. Una delle tecniche di hacking che vengono utilizzate per attaccare i database delle applicazioni web è l'SQL Injection, ovvero l'inserimento e l'esecuzione di codice SQL non previsto all'interno della pagina dinamica.

Struttura della tesi. In questa tesi si vuole proporre e realizzare un ambiente di sandboxing automatizzato per una applicazione di web security training. Nel Capitolo 1 viene descritta l'architettura classica di una web application e le principali tecnologie web. Nel Capitolo 2 sono introdotte le principali vulnerabilità che si possono trovare nelle applicazioni web. In particolare si analizzano in dettaglio le SQL Injection. Il Capitolo 3 descrive due strumenti di virtualizzazione: Vagrant e Docker. Sempre in questo capitolo viene introdotta l'applicazione di web training utilizzata per realizzare il sistema di sandboxing. Nel Capitolo 4 viene descritto il sistema di sandboxing che si vuole realizzare e vengono proposte due soluzioni basate sui due tool di virtualizzazione presentati. Nello stesso capitolo si analizza la differenza di prestazione tra le due soluzioni. Il Capitolo 5 riassume quanto realizzato e propone sviluppi futuri di questo sistema.

Capitolo 1

Web technologies

La rapida crescita di Internet ha portato negli ultimi anni alla creazione di molti servizi web che sono diventati in breve tempo parte della nostra vita quotidiana. Ogni giorno infatti ci capita di utilizzare applicazioni web per fare acquisti online, pagare le bollette o effettuare prenotazioni per le vacanze o per visite mediche. La ragione per cui le applicazioni web hanno avuto così tanta diffusione risiede nel fatto che per utilizzarla gli utenti hanno bisogno solamente di un browser web e di una connessione Internet. Tuttavia, parallelamente a questa facilità di utilizzo, la maggior parte delle applicazioni web presenta delle vulnerabilità di sicurezza che permettono a possibili attaccanti di sfruttarle per eseguire attacchi. Il risultato di tali attacchi porta alla perdita di confidenzialità, integrità e disponibilità delle informazioni a cui si vuole accedere sull'applicazione.

1.1 Architettura di una web application

In Figura 1.1 è mostrata una semplice architettura di sistema per una applicazione web. L'architettura è a tre livelli ed è composta da

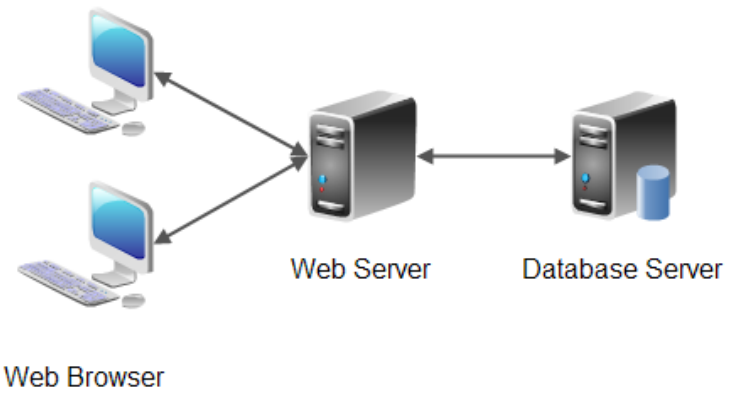

Figura 1.1: Architettura a 3 livelli per una applicazione web

- un **browser web** che ha la funzione di interfaccia con l'utente,
- un **application server** che gestisce la parte logica dell'applicazione e
- un **database server** che gestisce i dati persistenti.

L'*application server* riceve in input dei dati dagli altri due livelli: dal web browser riceve input dall'utente, mentre dal database riceve insiemi di risultati. Tipicamente alcuni di questi input vengono inviati in output agli altri due livelli sempre sotto forma di stringhe: verso il database queste stringhe sono query mentre verso il browser sono documenti HTML. L'application server si occupa di costruire il codice dinamicamente, in questo modo il codice dell'intera applicazione non è interamente in uno stesso punto. Il flusso di dati che si ha tra i tre livelli dell'architettura può portare al verificarsi di problemi di *input validation* per quanto riguarda l'application server. Questo infatti deve controllare o modificare le stringhe che riceve prima di processarle o di incorporarle nell'output da inviare agli altri livel-

li. Un errore nel controllo o nella sanitizzazione dell'input può portare alla compromissione della sicurezza dell'applicazione.

1.2 HTTP

L'HyperText Transfer Protocol (HTTP) è un protocollo a livello applicativo utilizzato come principale sistema di trasmissione di informazioni in una tipica architettura client-server. Solitamente HTTP sfrutta il protocollo TCP a livello di trasporto. La prima versione "sperimentale" di HTTP risale alla fine degli anni 1980, mentre la prima versione disponibile di questo protocollo è di alcuni anni dopo. Queste prime versioni del protocollo prevedevano la possibilità di avere una sola richiesta per connessione; questo poneva dei limiti evidenti, in particolare questo rendeva di fatto impossibile poter riutilizzare le connessioni disponibili, ospitare più siti web sullo stesso server e lasciava spazio alla mancanza di meccanismi di sicurezza adeguati. Per questi motivi alla fine degli anni 1990 il protocollo HTTP venne esteso all'attuale versione 1.1 che introdusse tra le novità la possibilità di effettuare richieste multiple utilizzando una sola connessione[1]. HTTP funziona come un protocollo di richiesta/risposta: il client effettua una richiesta e il server restituisce la risposta. In generale il client corrisponde ad un browser mentre il server è la macchina su cui risiede l'applicazione web. I messaggi in HTTP sono quindi di due tipi: messaggi di richiesta e messaggi di risposta. Il messaggio di richiesta è composto da quattro parti: request line, header, CRLF, body. La request line è composta a sua volta da un metodo, un URL e dalla versione del protocollo. Il metodo specifica il tipo di operazione che il client richiede

[1] https://tools.ietf.org/html/rfc2616

al server, sono definiti dallo standard e tra i più comuni troviamo: GET, POST, HEAD, PUT e DELETE.

L'URL identifica univocamente una risorsa sul server. Per quanto riguarda gli header della richiesta i più utilizzati sono due:

host – nome del server a cui si riferisce l'URL, obbligatorio nelle principali richieste di HTTP/1.1;

User agent – identifica il tipo di client e vi troviamo informazioni sul tipo e la versione del browser.

Il messaggio di risposta invece è di tipo testuale ed è composto da quattro parti come il messaggio di richiesta: status-line, header, CRLF, body. La status-line o linea di stato riporta un codice a 3 cifre chiamato appunto codice di stato appartenente ad una delle seguenti categorie:

- 1xx: Informational
- 2xx: Successful
- 3xx: Redirection
- 4xx: Client error
- 5xx: Server error

Gli header principali che troviamo nella risposta sono:

- **Server**: indica il tipo e la versione del server, è l'analogo di user-agent nella richiesta
- **Content-Type**: indica il tipo di contenuto restituito, questi tipo sono chiamati MIME ed hanno una codifica standard[2]. Ad esempio text/html oppure image/jpeg.

[2]https://www.ietf.org/rfc/rfc1521

Il body contiene il corpo del messaggio, ad esempio il file HTML richiesto.

Nel messaggio di richiesta il client può richiedere se utilizzare una comunicazione persistente o non persistente: il primo caso è quella di default della versione HTTP 1.1 in cui richiesta e risposta utilizzano la stessa connessione TCP mentre nel secondo caso per ogni richiesta e risposta vengono stabilite connessioni TCP dedicate.

Il protocollo HTTP differisce da altri protocolli di livello 7, come ad esempio FTP, per il fatto che le connessioni vengono generalmente chiuse una volta che una particolare richiesta o una serie di richieste correlate è stata soddisfatta. Questo comportamento rende il protocollo HTTP ideale per l'utilizzo Web in quanto permette di diminuire il numero di connessioni attive limitandole a quelle effettivamente necessarie aumentando quindi l'efficienza sia sul client che sul server. Tuttavia il fatto di essere *stateless* ci costringe a ricorrere all'utilizzo di meccanismi come i cookies per mantenere informazioni sulle richieste una volta che la connessione viene chiusa.

Il traffico su HTTP è in chiaro e per questo motivo sono state sviluppate diverse alternative per garantire diversi livelli di sicurezza: cifratura e integrità del traffico, autenticazione di client e server. Al momento la versione sicura utilizzata è HTTPS che si basa sull'utilizzo di un canale cifrato a livello di trasporto tramite SSL o TLS.

1.3 REST

Representional State Transfer (REST) è un insieme di principi e metodi di progettazione per sistemi ipermediali distribuiti. Questo stile di architettura è stato definito nel 2000 da Roy Fielding, uno degli autori delle specifiche di HTTP, nella sua tesi di dottorato [1]. L'analisi effettuata da Fielding

sulle risorse tecnologiche disponibili per la creazione di applicazioni distribuite porta ad affermare che, senza imporre alcun vincolo i sistemi tendono per loro natura ad evolvere in maniera entropica generando conseguenze negative, come l'elevato costo di realizzazione, mantenimento e sviluppo. Secondo Fielding, quindi, tali sistemi dovrebbero essere realizzati seguendo i principi REST per soddisfare specifiche caratteristiche, come ad esempio la scalabilità, le prestazioni, la modificabilità e la affidabilità.

I principi da seguire sono:

- **Architettura client-server**
- **Statelessness**
- **Cahcheability**
- **Layered system**
- **Code on demand**
- **Uniform interface**

Il principio fondamentale su cui si basa REST sono le risorse (fonti di informazioni), a cui si può accedere tramite un identificatore globale (un URI). Una *Risorsa* è tutto ciò che è abbastanza importante da poter essere descritta in maniera autonoma, ovvero se gli utenti possono creare un collegamento ipertestuale su di essa, recuperare o memorizzare una rappresentazione di essa o eseguire altre operazioni su di essa. Per utilizzare le risorse, le componenti di una rete (componenti client e server) comunicano attraverso una interfaccia standard (ad es. HTTP) e si scambiano rappresentazioni di queste risorse (il documento che trasmette le informazioni). Ogni risorsa deve essere identificata con un "nome", e in più deve essere

univoca. Questo permette di poter identificare la risorsa in maniera non ambigua e di raggiungerla da qualsiasi altro servizio. Il Nome può essere visto come l'indirizzo della risorsa e quindi la possibilità di dare alla risorsa la proprietà di essere indirizzabile, questa all'interno del servizio REST, viene definita mediante l'URI (Uniform Resource Identifier). Un client che usufruisce del servizio REST, manipola la risorsa connettendosi al server che la ospita e ne invia il percorso per raggiungerla.

Oltre a digitare gli URI per accedere alle risorse, è possibile interagire con esse per mezzo di azioni che vengono messe a disposizione del protocollo di comunicazione. A differenza della semplice consultazione, quando si eseguono questi metodi, la nostra operazione, modifica lo stato del server, ad esempio aggiungendo la nostra risorsa ad un determinato servizio. Il protocollo di trasmissione che viene utilizzato dai servizi REST è il protocollo standard HTTP. I metodi utilizzati sono quelli minimi messi a disposizione GET, POST, PUT e DELETE, mentre le altre operazioni disponibili dal protocollo possono essere reindirizzate a queste quattro.

1.3.1 NodeJS

Uno degli strumenti più utilizzati per implementare REST è il framework Node.js[3]. Node.js è un framework che permette di sfruttare V8, l'interprete JavaScript di Google, per creare applicazioni web e di rete veloci, compatte e affidabili. La principale caratteristica è che offre un modo per eseguire codice Javascript al di fuori del browser e lato server. Questo permette ad esempio di lavorare con i processi, accedere ai file oppure di utilizzare socket di rete con Javascript. Node.js è asincrono e questo gli permette di non bloccarsi mai durante l'esecuzione per aspettare una risposta. Esso infatti

[3]https://nodejs.org

si basa su dei gestori di eventi ed ha un approccio event-driven proprio come javascript che gli permette di essere scalabile e di gestire grandi carichi di lavoro. Ad esempio quando si effettua una query ad un database, invece di bloccarsi in attesa di una risposta, node.js registra una funzione callback che viene invocata quando il risultato del database è pronto, nel frattempo node.js può fare altre cose. Questo comportamento è realizzato tramite l'utilizzo di event loop che effettua due tipi di funzioni in un loop continuo: detection di eventi e trigger dei gestori di eventi. Questi event loop sono in esecuzione in un thread all'interno di un processo in modo tale per cui un event handler possa essere sempre in esecuzione in quel momento.

1.4 SQL

Structured Query Language (SQL) è un linguaggio standardizzato per database basati dul modello relazionale (RDBMS). Questo linguaggio è progettato per le seguenti operazioni:

- creare e modificare schemi di database
- inserire, modificare e gestire dati memorizzati
- interrogare i dati memorizzati
- creare e gestire strumenti di controllo ed accesso ai dati

A dispetto del nome non si tratta quindi solo di un semplice linguaggio di interrogazione, ma viene utilizzato per la creazione, la gestione e l'amministrazione del database. In base alle operazioni da eseguire possiamo dividere SQL in:

- Data Definition Language (DDL): permette di creare e cancellare database o di modificarne la struttura
- Data Manipulation Language (DML): permette di inserire, cancellare, modificare i dati
- Data Control Language (DCL): permette di gestire gli utenti e i permessi
- Query language (QL): permette di interrogare il database, cioè di leggere i dati.
- Device Media Control Language (DMCL): permette di controllare le memorie dove vengono memorizzati i dati.

SQL deriva dai concetti di modelli relazionali e calcolo relazionale su tuple. Nel modello relazionale una tabella è un insieme di tuple mentre in SQL le tabelle e i risultati di una query sono liste di righe: la stessa riga può essere presente più volte e l'ordine delle righe può essere specificato nelle query. La sintassi di SQL è definita dall'ISO/IEC SC 32 e gli elementi che la compongono sono:

- Clauses, che sono le componenti principali di statement e query, in alcuni casi sono opzionali
- Espressioni: possono produrre valori scalari o tabelle i dati
- Predicati: specificano condizioni che possono essere valutate tramite logica SQL (true/false/unknown) o Booleana e sono utilizzati per limitare gli effetti degli statement e delle query oppure per cambiare il flow del programma.

- Queries: restituiscono dati basandosi su criteri specifici.
- Statements: possono avere effetti persistenti sullo schema de database e sui dati, possono controllare transazioni, flow del programma connessioni e sessioni

Capitolo 2

Vulnerabilità

Le applicazioni Web presentano, per loro natura, delle vulnerabilità che non sono solo frutto di errori di programmazione da parte degli sviluppatori, comunemente chiamati bug. Spesso le vulnerabilità nascono già nella fase di progettazione dell'applicazione stessa. La gestione non corretta degli input, ad esempio, difficilmente è dovuta ad errori di programmazione ma spesso sono introdotte da scelte sbagliate durante la fase di progetto. Tra le vulnerabilità più diffuse infatti, troviamo quelle relative alla validazione degli input ed in testa alla classifica troviamo le *injection flaws*[2].

2.1 Vulnerabilità di Input Validation

Questo tipo di vulnerabilità è tra i più sfruttati dagli attaccanti per effettuare attacchi di *code injection*.

Le due classi più importanti di vulnerabilità riguardanti l'*input validation* sono i Cross-Site scripting (XSS) e le SQL injection delle quali si parlerà in maniera approfondita nel Capitolo 2.3. Per entrambe le classi l'obiettivo dell'attaccante è quello di far produrre all'application server un

output deciso dallo stesso attaccante e non previsto dal programmatore dell'applicazione. In particolare nel caso di XSS l'attaccante farà produrre all'application server un documento HTML. Tramite l'utilizzo di SQL injection invece l'attaccante sarà in grado di alterare le query inviate al database.

Questi due attacchi sono resi possibili dal fatto che l'applicazione costruisce i documenti HTML e le query al database attraverso una manipolazione delle stringhe a basso livello e tratta gli input non attendibili dell'utente come elementi lessicali isolati. Questo comportamento è molto comune specialmente nei linguaggi di scripting come PHP che utilizzano stringhe come rappresentazione di default per dati e codice. Le vulnerabilità di input validation rappresentano la maggior parte delle vulnerabilità di cui possono soffrire le web application. I dati che si ricevono da una entità esterna o da un client infatti non dovrebbero mai essere considerati fidati. La prima regola da seguire infatti è quella di considerare tutti gli input come malevoli o citando Michael Howard "All input is evil"[3].

Tuttavia sfortunatamente molto spesso le applicazioni sono complesse e hanno un gran numero di *entry point*; questo rende molto difficile per lo sviluppatore seguire questa regola. Proprio per questo motivo, sebbene negli ultimi anni si sia riscontrato un calo della diffusione di questo tipo di vulnerabilità nelle web application, non sono mancati casi in cui si sono trovate vulnerabilità a XSS e SQL injection riguardanti anche web application molto note.

2.2 injection Flaws

Le *injection flaws* permettono agli attaccanti di trasmettere codice malevolo ad un altro sistema attraverso una applicazione. Questo tipo di attacchi includono chiamate al sistema operativo tramite l'utilizzo di *system calls*, l'utilizzo di programmi esterni tramite comandi via *shell* e chiamate ai database via SQL (SQL injection). Il fatto che una applicazione sia stata progettata in modo non conforme può permettere ad un attaccante di andare ad inserire al suo interno interi *script*. Inoltre molte applicazioni web utilizzano funzionalità del sistema operativo e programmi esterni per eseguire particolari azioni come, l'invio di email. Quando una applicazione web passa informazioni tramite una richiesta HTTP come parte di una richiesta esterna questa deve essere gestita con attenzione. Se ciò non avviene, l'attaccante può iniettare caratteri speciali o comandi malevoli nelle informazioni e l'applicazione web la passerà ciecamente al sistema esterno per essere eseguiti.

2.3 SQL injection

Gli attacchi di code injection riguardano un campo molto flessibile e sottoposto a continue introduzioni di nuovi elementi, pertanto è difficile classificare in maniera rigorosa le tecniche di SQL injection. Spesso, inoltre, un attaccante non utilizza una sola tecnica per raggiungere i suoi scopi ma una combinazione di queste[4]. Si deve infine considerare anche il fatto che per ogni tipo di attacco vi sono molteplici variazioni possibili.

Una possibile classificazione è la seguente [5]:

Tautologie Questo tipo di attacco ha come scopo quello di inserire del codice all'interno di uno o più statement condizionali in modo che

questi vengano valutati sempre a vero. Le conseguenze di questo attacco dipendono dai risultati della query; questa tecnica viene utilizzata principalmente per bypassare le pagine di autenticazione o per estrarre dati.

Query non corrette Solitamente questo è un attacco preliminare utilizzato da un attaccante per ottenere informazioni sul tipo di database e sulla sua struttura. Queste informazione vengono poi utilizzate in attacchi successivi e più complessi. La vulnerabilità in questo caso è rappresentata dalla pagina di errore di default che viene restituita da parte dell'applicazione la quale solitamente è troppo descrittiva e restituisce informazioni poco utili per un utente medio ma che per un attaccante rappresentano invece informazioni molto importanti.

Union Query Questo attacco sfrutta un parametro vulnerabile per cambiare i dati restituiti da una certa query. L'attaccante con questa tecnica può forzare l'applicazione a restituire un data da una tabella diversa rispetto a quella prevista originariamente dallo sviluppatore.

Stacked Queries/ Piggy-Backed Queries In questo tipo di attacco si sfrutta la possibilità di inserire query addizionali nella query originale. Rispetto agli altri tipi di attacco in questo caso non viene modificata la query originale ma semplicemente se ne aggiungono altre in serie. Il risultato è che il database eseguirà sia la query originale che quelle aggiunte dall'attaccante. Questo tipo di attacco è molto pericoloso in quanto l'attaccante non è legato ad una query predefinita ma può inserire qualsiasi tipo di comando SQL. Il caso più noto è il drop tables.

Stored Procedures Attacchi di questo tipo cercano di eseguire le stored procedure (mettere nota foot per spiegare cosa sono) presenti nel database, specialmente quelle che interagiscono con il sistema operativo.

Inferenza Questo tipo di attacco va a modificare la query in modo da ottenere dal database una risposta di tipo TRUE/FALSE in base ai dati presenti sul database. Questa tecnica è simile a quella delle query non corrette con la differenza che questa è utilizzabile nel caso in cui non vengano restituite pagine di errore da parte dell'applicazione. L'attaccante riesce ad identificare lo schema del database o anche ad estrarre dati osservando le variazioni nel comportamento dell'applicazione. Ci sono due tipi principali di attacchi: le *Blind injection* e i *Timing Attacks*. Il primo tipo si basa su statement che restituiscono valore true/false, mentre il secondo sfrutta il time delay di risposta del database e le query utilizzate sono solitamente nella forma if/then.

Di seguito vengono mostrati nel dettaglio alcuni esempi di possibili attacchi di SQL injection che possono essere testati utilizzando l'applicazione cyber-gym della quale parleremo in dettaglio in Sezione4.1. In particolare vengono analizzati un attacco di tipo tautologico e un attacco stacked query.

2.3.1 Login bypass

La tecnica per il bypass del login è senza dubbio una delle tecniche di SQL injection più popolari. In questa sezione vediamo un esempio di come un attaccante possa superare una form di login.

La form di login che prendiamo in considerazione è quella classica. Si hanno semplicemente due campi di input per consentire all'utente di inserire username e password. Il codice di backend che genera la query per la validazione di username e password e l'autenticazione dell'utente è il seguente:

```
1  # login.php
2  <?php
3  include('../db/mysql_credentials.php');
4  $con = new mysqli( $mysql_server, $mysql_user,
       $mysql_pass, $mysql_db );
5  if ($con->connect_error) die ("Connection failed: " .
       $con->connect_error);
6  $email = $_POST['email'];
7  $pass = $_POST['pass'];
8  $query = "SELECT * FROM accounts WHERE email='$email'
       AND password=SHA('$pass')";
9  $result = $con->query($query);
10 if($result->num_rows > 0) {
11   $row = $result->fetch_assoc();
12   $first_name = $row["first_name"];
13   $last_name = $row["last_name"];
14   echo "Login successful. Welcome $first_name
         $last_name!";
15 } else {
16   echo "Wrong username or password";
17 }
18 $con->close();
```

Il codice appena visto è corretto sia dal punto di vista sintattico che dal punto di vista logico. Infatti se un utente prova ad effettuare il login con le proprie credenziali corrette lo script gli garantirà l'accesso, al contrario utilizzando credenziali non corrette l'accesso verrà negato. Tuttavia lo script presenta una vulnerabilità di tipo code injection, in questo caso SQL injection. Un attaccante infatti può superare la form di login andando ad inserire nei campi di input delle stringhe particolari che permettono la generazione di query SQL che risultano corrette sintatticamente ma non previste dallo sviluppatore. Un possibile esempio di attacco è il seguente:

```
#Username e password malevola inserite dall'
    attaccante
username: admin
password: wrongpassword' OR 'a'='a

#Query generata
SELECT * FROM users WHERE username='admin' AND
    password='wrongpassword' OR 'a'='a'
```

Per bypassare il login e accedere all'area riservata l'attaccante costruisce un segmento di SQL che inserito come valore di password andrà a modificare la clause WHERE e rendere sempre vero il valore ritornato dalla query. Nell'esempio mostrato sopra vengono mostrati i valori inseriti nelle form di input dall'attaccante e la query che questi vanno a formare. A causa della proprietà di precedenza degli operatori booleani, la condizione in AND viene valutata per prima. In seguito viene valutato l'operatore OR il quale rende sempre true la condizione in WHERE. Infatti la condizione sarà vera per tutte le righe della tabella user. Questo implica che lo username fornito viene ignorato e l'attaccante effettuerà il login con le credenziali del primo

utente nella tabella user. Questo vuol dire inoltre che per l'attaccante non è necessario conoscere uno username per accedere al sistema ma la query gliene fornirà direttamente uno.

Anche il campo dell'username può avere la stessa vulnerabilità del campo password. Anch'esso quindi può essere soggetto ad exploit per accedere al sistema; un esempio è il seguente:

```
#Username e password malevoli inseriti dall'
    attaccante
admin' --
anypassword

#Query generata
SELECT * FROM users WHERE username='admin' -- AND
    password='anyPassword'
```

In questo modo inoltre l'attaccante può scegliere con quale account ottenere l'accesso dopo aver effettuato il bypass del login. Nell'esempio viene utilizzato il metodo del commento per eliminare la seconda parte della query e accedere come admin.

2.3.2 Stacked queries

Questo tipo di attacco consiste nello sfruttare le vulnerabilità di una qualsiasi form di input, abbia essa lo scopo di effettuare il login o una semplice ricerca, per andare ad eseguire più statement in una sola query. Infatti terminando la query originale e aggiungendone subito dopo un'altra è possibile per l'attaccante modificare i dati ed eseguire stored procedure. Sempre facendo riferimento all'applicazione cyber-gym vediamo un esem-

pio di questo attacco. In figura vediamo una semplice form che ci permette di effettuare una ricerca all'interno del nostro database e ci restituisce un risultato. Lo script di backend che gestisce la richiesta è il seguente:

```
#stacked_queries.php
<?php
include('../db/mysql_credentials.php');
if (!isset($_GET['email'])) {
  echo "<form action=''>";
  echo "<label for='email'>Search by email</label><
     input name='email'>";
  echo "</form>";
} else {
  $email = $_GET['email'];

  $con = new mysqli( $mysql_server, $mysql_user,
     $mysql_pass, $mysql_db );
  $query = "SELECT * FROM accounts WHERE email='
     $email'";
  var_dump($query);
  $con->multi_query($query);
  do {
    if ($result = $con->store_result()) {
      echo "<table>";
      echo "<tr><th>First Name</th><th>Last Name</th
         ><th>Email</th></tr>";
      while( $row = $result->fetch_assoc() ) {
        $first_name = $row['first_name'];
```

```
            $last_name = $row['last_name'];
            $email = $row['email'];
            echo "<tr><td>$first_name</td><td>$last_name
                </td><td>$email</td></tr>";
        }
        echo "</table>";
    }
  } while (@$con->next_result());
  $con->close();
}
```

Anche in questo caso il codice porta a termine il suo compito in maniera corretta ma è soggetto ad una vulnerabilità per quanto riguarda l'injection di codice. In maniera analoga al caso di login bypass un attaccante può andare ad inserire una stringa non prevista dallo sviluppatore ed eseguire operazioni non volute. Un esempio è il seguente:

```
#Input dell'attaccante nella form
admin'; DROP TABLE users;--


#Query generata
SELECT * FROM users WHERE username='admin'; DROP
    TABLE users; -- ';
```

In SQL il punto e virgola (semicolon) indica che è stata raggiunta la fine di uno statement. Il codice mostrato sopra permette di eseguire più statement nella stessa chiamata al server del database. In questo caso un utente viene ritornato dal primo statement e tutto il database viene cancellato dal secondo. Al contrario degli attacchi UNION che vedremo successivamen-

te i quali sono limitati a statement SELECT, le stacked queries possono essere usate per eseguire qualsiasi statement o procedura SQL. Tuttavia è importante sottolineare come questo tipo di attacco non funziona in tutte le situazioni. La maggior parte delle volte infatti questo attacco non è supportato dalle API o dall'engine del database. Questo attacco può essere utilizzato anche per estrarre dati tramite l'utilizzo di statement SELECT, tuttavia non è detto che il codice permetta di gestire risultati ritornati da più query con il risultato che le query successive vengano ignorate o provochino errori. In questo caso risultano più utili gli attacchi UNION.

Capitolo 3

Strumenti di virtualizzazione

La virtualizzazione è il processo per il quale è possibile simulare il comportamento di una macchina fisica tramite software. Ad oggi la virtualizzazione può essere sfruttata sia a livello software che a livello hardware. Tra gli impieghi possibili il più utilizzato è senza dubbio la virtualizzazione dei sistemi operativi, ma negli ultimi anni è in forte crescita la virtualizzazione dei server. In questo capitolo vengono presentati due tool che permettono di realizzare la ambienti virtuali, ciascuno dei quali presenta caratteristiche differenti che verranno analizzate in dettaglio.

3.1 Vagrant

3.1.1 Cos'è Vagrant

Vagrant[1] è un software open-source per la realizzazione e la configurazione di ambienti di sviluppo su macchine virtuali [6]. Tramite l'utilizzo di Vagrant è quindi possibile per gli sviluppatori ridurre i tempi per la creazione,

[1] https://www.vagrantup.com/

la preparazione e la gestione dell'ambiente di sviluppo e allo stesso tempo incrementare la produttività.

Tramite un semplice comando, infatti, Vagrant è in grado di eseguire le seguenti operazioni:

- creare una macchina virtuale basata su un sistema operativo scelto dall'utente;
- modificare le proprietà fisiche della macchina virtuale: ad esempio il quantitativo di memoria RAM o il numero di CPU;
- stabilire le interfacce di rete per accedere alla macchina virtuale dal PC dell'utente su cui viene avviata la macchina virtuale o da un altro device presente sulla stessa rete o anche da un'altra macchina virtuale;
- creare le cartelle condivise tra la macchina virtuale e il PC dell'utente; quest'ultimo andrà a creare e modificare sul proprio PC i file in queste cartelle e le modifiche risulteranno immediatamente visibili sulla macchina virtuale;
- effettuare il *boot* della macchina;
- impostare un *hostname* della macchina;
- effettuare il *provisioning* del software sulla macchina attraverso l'utilizzo di script.

Tutte queste operazioni possono essere completate in pochi minuti. Una volta che Vagrant ha settato la macchina virtuale l'utente ha a disposizione un ambiente di sviluppo completo e sandboxato. Grazie all'utilizzo della rete e delle cartelle condivise create l'utente può continuare ad usare il

proprio editor preferito e il proprio browser per sviluppare e testare le proprie applicazioni, mentre il codice viene eseguito sulla macchina virtuale. Vagrant inoltre gestisce l'intero ciclo di vita della macchina al posto dell'utente. Mentre la macchina virtuale è in esecuzione, infatti, possono essere eseguite le seguenti operazioni:

- Connettersi tramite ssh alla macchina virtuale
- Spegnere la macchina
- "Distruggere" la macchina virtuale eliminando il tutti i suoi dati e metadati
- Sospendere l'esecuzione della macchina
- Salvare lo stato della macchina per poter essere distribuita ad altri sviluppatori

Oltre a queste operazioni Vagrant può fare molto altro, infatti grazie alla sua natura open-source esiste possibilità di integrare dei plug-in per ampliare le funzionalità di Vagrant in base alle necessità dell'utente.

3.1.2 Perché utilizzare Vagrant

La scelta di utilizzare uno strumento come Vagrant è motivata dal fatto che configurare gli ambienti di sviluppo in modo standard non è impresa semplice e il più delle volte è estremamente time-consuming. Per sviluppare applicazioni web è necessario installare e configurare sulla macchina che si vuole utilizzare diversi software come ad esempio Apache, MySQL, etc. Al giorno d'oggi le applicazioni web sono costituite da diverse parti e si rende necessario l'utilizzo di diversi linguaggi di programmazione, da

JavaScript a PHP a Python. Allo stesso modo per i database esistono diverse soluzioni in base all'utilizzo che se ne desidera fare. Oltre a ciò sono disponibili molti web server e servizi di backend ciascuno dei quali con il proprio caso d'uso. Le possibili combinazioni di tutte queste tecnologie sono molteplici e avere ognuna di esse installata e configurata alla perfezione localmente è molto complicato. Infatti ogni software deve essere installato manualmente e spesso certi software potrebbero non essere compatibili con il sistema operativo utilizzato. Una volta installato il software deve essere configurato e questo spesso risulta più complicato dell'installazione stessa. Esso infatti deve essere configurato al meglio per ottimizzare il processo di production, infatti un software non configurato bene può funzionare senza problemi in fase di developement ma non in production. Il setting manuale degli ambienti di sviluppo infatti è uno delle cause principali del bug "sul mio PC funziona". Idealmente la configurazione dello sviluppo deve rispecchiare identicamente quella di production. Inoltre lo sviluppo di progetti multipli risulta alquanto difficoltoso rasentando l'impossibile in alcuni casi in quanto ogni progetto necessita una differente configurazione dei servizi da esso richiesti o addirittura un sistema di backend completamente diverso. Il risultato che si ottiene è un ambiente configurato in maniera poorly o addirittura un sistema che possiede servizi che la web app individualmente non necessitano. In sostanza un ambiente la cui confusione tende ad aumentare di pari passo all'utilizzo. Un altro aspetto da considerare è la difficoltà nel sincronizzare gli ambienti di sviluppo tra i membri dello stesso team o con l'aggiungere un nuovo membro al team di sviluppo. Poiché ogni sviluppatore è responsabile di un ambiente di sviluppo separato è facile che questi vadano fuori sincronia velocemente. I nuovi membri del team sono responsabili personalmente del setting del proprio ambiente

in maniera completa. Ciò che si ottiene da questa situazione è confusione e una lunga attesa prima che un nuovo membro del team possa effettivamente cominciare a lavorare al progetto. Infine per molti sviluppatori è difficile usare diversi sistemi operativi e su alcuni di questi non sempre è possibile utilizzare il software che si vuole sfruttare per un certo progetto. Alcune volte può addirittura capitare che un certo software non sia neppure compatibile con un particolare sistema operativo. Per questo motivo gli sviluppatori tendono ad utilizzare il sistema meglio documentato per impostare il proprio ambiente di sviluppo.

Vagrant offre una soluzione a tutti questi problemi. Il set up dell'ambiente di sviluppo può infatti essere automatizzato tramite l'utilizzo di shell script o di software di configurazione. Vagrant permette inoltre di lavorare con lo stesso sistema operativo che viene utilizzato in produzione indipendentemente che sulla macchina dello sviluppatore sia presente Linux, Windows o Mac OS. Vagrant infatti inserisce l'ambiente di sviluppo all'interno di una macchina virtuale. anche lavorare su più progetti risulta più semplice poiché ogni progetto avrà una macchina virtuale dedicata. infine anche il lavoro in team viene reso più semplice grazie alla possibilità di condividere l'immagine della macchina virtuale, così come poter mettere subito a disposizione l'ambiente di sviluppo per un nuovo membro del team tramite un semplice comando.

3.1.3 Vagrantfile e componenti

Vagrantfile

Il Vagrantfile è un semplice file di testo che Vagrant legge in modo tale da determinare quali operazioni devono essere eseguire per creare l'ambiente

di lavoro corrispondente[5]. Questo file descrive quale sistema operativo deve essere avviato per l'ambiente virtuale, quali saranno le proprietà fisiche della macchina virtuale, quale software dovrà essere installato su tale macchina e quali metodi di accesso ad essa dalla rete dovranno essere disponibili. Vagrant è configurato sul singolo progetto e ogni progetto ha un proprio ambiente di lavoro isolato. Un progetto è identificato dall'esistenza di un file chiamato appunto "Vagrantfile". Ciascun progetto ha un unico Vagrantfile corrispondente. Questo file è destinato ad essere posizionato nella versione di controllo dove all'occorrenza un altro membro del team potrà facilmente ricreare da esso l'ambiente di lavoro senza dover configurare nulla e risparmiando molto tempo. I Vagrantfile sono pensati per essere raramente modificati, in quanto l'impostazione di base dell'ambiente di sviluppo rimane relativamente stabile rispetto al codice software che viene sviluppato. La maggior parte del costo si ha quindi all'inizio mentre si guadagna in stabilità e flessibilità.

Il Vagrantfile è scritto in un linguaggio di programmazione chiamato Ruby. Ruby è un linguaggio di programmazione orientata agli oggetti; la sua semantica è simile ad altri linguaggi e non è necessario conoscere a fondo questo linguaggio per comprendere e realizzare un Vagrantfile.

Di seguito possiamo vedere un estratto esempio di un semplice Vagrantfile contenente alcune istruzioni principali:

```
1 Vagrant.configure("2") do |config|
2     config.vm.box = "scotch/box"
3     config.vm.network "private_network",
                        ip: "192.168.33.10"
4     config.vm.hostname = "scotchbox"
5     config.vm.synced_folder ".", "/var/www",
```

```
                    :mount_options => ["dmode=777", "
        fmode=666"]
6       config.vm.provision: "shell", path: "provision.sh
        "
7   end
```

Il blocco di configurazione di Vagrant inizia alla linea 1 tramite lo statement *do* impostando una variabile *config* valida per l'intera durata del blocco. Il valore *2* tra le parentesi indica che si sta usando la versione 2 di configurazione. All'interno del blocco di configurazione troviamo tre chiamate a funzione (linee 3, 5 e 6) e due assegnamenti di variabile (linee 2 e 4). Infine all'ultima linea troviamo la chiusura tramite lo statement *end* del blocco di configurazione.

Come si vede con poche righe di codice e usando semplici combinazioni di assegnamento a variabili, chiamate funzione e blocchi di configurazione è possibile comporre un Vagrantfile che ci permette di realizzare un semplice e completo ambiente di sviluppo.

Componenti nel dettaglio

Compresa la sintassi del Vagrantfile possiamo ora analizzare che cosa vogliono dire le linee di codice che abbiamo visto. Sempre con riferimento al codice riportato in figura andiamo a vedere nel dettaglio quali operazioni vengono effettuate in particolare dalle funzioni utilizzate nel Vagrantfile. Nel caso del Vagrantfile riportato ed utilizzato per il progetto abbiamo quattro funzioni che corrispondono ad altrettanti importanti aspetti riguardanti Vagrant, ovvero: box, network, synced folders e provisioning.

Box. La linea 2 specifica il nome del box richiesto per il nostro progetto. Un box è un'immagine base di una macchina virtuale che contiene un sistema operativo già installato. Realizzare una macchina virtuale partendo da zero infatti risulta piuttosto impegnativo sia dal punto di vista delle risorse sia dal punto di vista del tempo. Utilizzando questi template il tempo impiegato per la realizzazione della macchina risulta decisamente minore. Vagrant mappa il nome del box al corrispettivo box presente sul sistema e nel caso in cui questo non sia presente provvede a scaricarlo. Vagrant usa i box solo come immagine da copiare all'interno della macchina virtuale, per cui il box originale non verrà mai modificato e potrà essere riusato come base per realizzare altre macchine per altri progetti. Il box che è stato utilizzato per questo progetto si chiama Scotchbox ed è l'immagine di Ubuntu 14.04LTS[7].

Network. La chiamata a funzione successiva nel Vagrantfile di esempio è la funzione network che, come suggerisce il nome, serve a configurare alcune impostazioni relative alla connessione di rete della macchina. Vagrant configura in maniera automatica diverse opzioni per il networking con la macchina virtuale, questo permette di utilizzare lo stesso Vagrant per comunicare con la macchina creata. Questa caratteristica di Vagrant è ovviamente molto importante per ambienti come quelli di progettazione web, infatti questo rende possibile per gli sviluppatori utilizzare il proprio browser e tool di sviluppo per accedere al progetto mentre il codice dell'applicazione web e tutte le sue dipendenze vengono eseguite in isolamento all'interno della macchina virtuale. Nel Vagrantfile esaminato si ha un esempio di Host-Only networking (private networking), ossia viene creata una rete privata tra il proprio host e la macchina ospite su quel-

l'host. Poiché questa è una nuova rete, ad essa va assegnato uno spazio di indirizzamento IP in modo tale che la macchina ospite ottenga un proprio IP. Vagrant supporta questo tipo di configurazione andando a specificare un indirizzo IP statico per la macchina. Successivamente si potrà effettuare l'accesso alla macchina usando direttamente l'indirizzo IP assegnato. Configurando in questo modo la rete si ottengono molti vantaggi, in particolare questo tipo di rete è sicura in quanto computer esterni non potranno accedere ai servizi di rete che andrò ad utilizzare. Inoltre tramite la configurazione Host-Only è possibile far comunicare tra loro diverse macchine virtuali appartenenti alla stessa rete tramite la conoscenza reciproca degli indirizzi IP assegnati ad esse. Questo può essere utile se si vuole separare più servizi su diverse macchine virtuali come ad esempio avviene per web server e database. Oltre a comunicare tra loro le macchine virtuali possono anche comunicare con l'host, utile nel caso in cui si voglia accedere a servizi presenti sull'host. L'unico svantaggio che si può avere utilizzando questo tipo di configurazione di rete si ha nel caso in cui si voglia condividere in tempo reale con gli altri membri del team di sviluppo il proprio lavoro a causa dell'isolamento che si viene a creare.

Alla linea 3 del Vagrantfile che è riportato in precedenza, troviamo la direttiva *config.vm.network* che specifica che stiamo configurando la rete della macchina virtuale. Il primo argomento è *private_network* il quale indica che la rete creata sarà privata o Host-Only, mentre il secondo argomento è l'indirizzo IP statico che verrà assegnato alla macchina virtuale. L'indirizzo IP può essere qualunque, anche se è consigliabile utilizzare un *range* privato e riservato come da specifiche RFC. Se dalla macchina virtuale si vuole accedere all'*host*, l'indirizzo di quest'ultimo sarà lo stesso indirizzo della macchina virtuale, con la sola differenza che l'ultima parte

dell'indirizzo (l'ultimo ottetto) sarà sempre 1. Ad esempio, se assegno alla macchina virtuale l'indirizzo 192.168.33.10, allora l'indirizzo dell'host per quella rete sarà 192.168.33.1.

Vagrant permette altri due tipi di connessione di rete: *forwarded ports* e *bridged networking*. Il primo permette di aprire alcune porte da utilizzare per accedere ai servizi sulla macchina virtuale senza utilizzare un indirizzo IP. Si può far lavorare le porte sia per connessioni TCP che per UDP, inoltre Vagrant offre al suo interno un servizio di *collision detection* per le porte. Il secondo tipo di configurazione assegna un indirizzo IP alla macchina virtuale in modo tale da farla sembrare una vera macchina fisica. Questi tre tipi di configurazioni sono componibili tra loro per soddisfare tutte le esigenze degli sviluppatori.

Synced folders. Alla linea 5 si ha la direttiva relativa alle *synced folders*. Questo permette a Vagrant di sincronizzare una cartella tra la macchina *host* e la macchina *guest*, in questo modo lo sviluppatore può continuare a lavorare sui file relativi al proprio progetto sulla propria macchina e di utilizzare le risorse della macchina virtuale per compilare ed eseguire il progetto. Di default Vagrant condivide la cartella del progetto in */vagrant* ma tramite la funzione di configurazione del Vagrantfile *config.vm.synced_folder* possiamo specificare tramite il primo parametro l'indirizzo alla cartella sulla macchina host che vogliamo condividere, mentre con il secondo parametro si indica il percorso assoluto dove andrò ad inserire la cartella nella macchina guest.

Provisioning. L'ultimo comando presente nel Vagrantfile in figura è relativo al *Provisioning*. Con il termine Provisioning viene indicato il problema di installare del software su di un sistema avviato. Questo è spesso

fatto tramite l'utilizzo di shell scripts, sistemi di gestione di configurazione o manualmente tramite linea di comando. In generale i *box* base utilizzati con Vagrant sono abbastanza spogli e contengono la minima quantità di software necessari per l'utilizzo. Vagrant supporta il provisioning automatizzato e all'avvio della macchina verrà installato su di essa il software di cui lo sviluppatore avrà bisogno per il suo ambiente. Questo può essere fatto utilizzando shell scripts, Chef[2], Puppet[3] o eventualmente altri tool aggiungibili tramite *plug-in* a Vagrant. I vantaggi che si ottengono dall'utilizzare il provisioning automatizzato sono tre: la semplicità d'utilizzo, la riusabilità e migliorare la parità tra sviluppo e produzione. Nel Vagrantfile in figura viene utilizzato il provisioning tramite shell script. Il comando utilizzato è *config.vm.provision* indicando come primo parametro il tipo di provisioning e come secondo parametro in questo caso il path dello script da utilizzare. Similmente a quanto avviene nella configurazione della rete anche per il provisioning è possibile utilizzare più di un metodo contemporaneamente se lo sviluppatore lo richiede.

3.1.4 Vagrant up ed altri comandi

A questo punto il Vagrantfile è completo ed è possibile realizzare l'ambiente di sviluppo in Vagrant. Questo viene fatto eseguendo semplicemente tramite terminale il seguente comando:

```
vagrant up
```

Una volta completate le operazioni e le configurazioni presenti nel Vagrantfile si avrà in esecuzione una macchina virtuale realizzata in modo automatico con un sistema operativo scelto dallo sviluppatore e con tutto

[2]https://www.chef.io/

[3]https://puppet.com/

il software di cui ha bisogno per lo sviluppo del proprio progetto già installato e pronto all'uso. Vagrant esegue la macchina virtuale di default senza una interfaccia grafica ma tra i processi in esecuzione sulla macchina comparirà il processo VBoxHeadless. Una volta creato l'ambiente di sviluppo Vagrant e mandato in esecuzione è possibile interagire con esso tramite diversi comandi che ci permettono di controllare lo stato della macchina virtuale, condividere dei file, comunicare con essa tramite la rete, accedere alla macchina virtuale oppure distruggere l'ambiente creato per ricrearlo nuovamente o crearne uno nuovo.

Poiché Vagrant esegue le macchine virtuali senza interfaccia grafica è facile dimenticarsi qual'è lo stato attuale dell'ambiente, per questo un comando molto utile è `vagrant status`:

```
$ vagrant status
Current machine states :
machine1          running (virtualbox)
machine2          not created (virtualbox)
machine3          poweroff (virtualbox)
machine4          suspended (virtualbox)
```

Listing 3.1: esempio su terminale

Come si può notare, con questo comando Vagrant restituisce lo stato corrente delle macchine che vengono identificate tramite il nome a loro assegnato nel Vagrantfile. Ogni macchina può essere in uno dei seguenti stati: running, suspended, saved, not created. Nel caso in cui si abbiano più macchine, è possibile controllare lo stato di una singola macchina aggiungendo al comando il nome della macchina della quale si vuole conoscere lo stato.

L'accesso alla macchina viene eseguito tramite il comando:

```
vagrant ssh <nome_macchina>
```

Questo comando gestisce la connessione ssh tra host e guest restituendo all'utente un prompt della macchina. Una volta effettuato l'accesso alla macchina è possibile svolgere qualsiasi operazione: installare software, modificare file, cancellare il filesystem, ecc.

L'esecuzione della macchina può essere terminata in tre modi diversi in base alle proprie esigenze.

Con il comando `vagrant suspend` verrà salvato lo stato corrente della macchina e ne verrà sospesa l'esecuzione.

```
vagrant suspend
```

In un secondo momento, la macchina può essere avviata nuovamente a partire dallo stato in cui si trovava al momento della sospensione. Questo permette di riprendere il lavoro molto velocemente, mantenendo intatti i servizi web, i database e la memoria della macchina. Uno svantaggio di questa scelta è l'occupazione di spazio su disco, in quanto oltre allo spazio occupato dalla macchine virtuale, che si aggira intorno ai 2 GB, si deve considerare anche uno spazio aggiuntivo in cui si andrà a salvare il contenuto della RAM della macchina virtuale. Dopo la sospensione, la macchina virtuale non consumerà più CPU e RAM, ma continuerà ad occupare spazio su disco; eseguendo il comando status quando la macchina è sospesa questo ci restituirà il valore saved.

Utilizzando il comando `vagrant halt` invece si effettua un vero e proprio spegnimento della macchina virtuale.

```
vagrant halt
```

Il beneficio di utilizzare lo spegnimento della macchina è che la sulla macchina guest non viene occupata memoria extra su disco, a differenza della sospensione, in quanto lo stato attuale della macchina virtuale in questo caso non viene salvato. A differenza della sospensione infatti la memoria RAM della macchina virtuale non viene preservata. La macchina virtuale continua comunque ad essere presente in memoria e ad occupare spazio su disco. In seguito al comando di `halt` la macchina si troverà nello status poweroff.

Il comando vagrant destroy infine, oltre a spegnere la macchina virtuale provvede a rimuoverne qualsiasi traccia dalla macchina host.

```
vagrant destroy
```

Il beneficio di questo comando è che viene ripristinato lo stato della macchina fisica come se non fosse mai stato eseguito nessun comando `vagrant up`. Tuttavia l'esecuzione successiva di un comando vagrant up richiederà una riconfigurazione completa dell'ambiente di sviluppo a partire dall'import del box scelto. Questo ovviamente comporta tempi di attesa molto più lunghi rispetto alla ripresa da una sospensione o da uno spegnimento della macchina.

In tutti e tre i casi per riavviare la macchina è sufficiente dare il comando `vagrant up`.

3.1.5 Vagrant multimachine

Una caratteristica molto importante di Vagrant è la possibilità di creare un ambiente di sviluppo con la presenza di più macchine virtuali, questo ad esempio può avvenire per lo sviluppo di applicazioni web che spesso sono composte da più parti distinte chiamate servizi. Vagrant permette di

realizzare ambienti di sviluppo di questo tipo in maniera molto efficiente tramite una feature chiamata multimachine enviroment che permette di realizzare più macchine virtuali utilizzando un solo Vagrantfile:

```
1  Vagrant.configure("2") do |config|
2      config.vm.box = "scotch/box"
3  config.vm.define "web", primary: true do |web|
4      web.vm.network "private_network",
                              ip: "192.168.33.10"
5      web.vm.hostname = "scotchbox"
6      web.vm.synced_folder ".", "/var/www", :
           mount_options => ["dmode=777", "fmode=666"]
7      end
8  config.vm.define "db" do |db|
9      db.vm.network "private_network",
                               ip: "192.168.33.11"
10     db.vm.synced_folder ".", "/var/www", :
           mount_options => ["dmode=777", "fmode=666"]
11     db.vm.provision :shell, path: "db_provision.sh"
12     end
13 end
```

In questo esempio vengono realizzate due macchine virtuali separate che assumono rispettivamente il ruolo di web server e di database tramite le quali lo sviluppatore può simulare ad esempio una failure della rete e osservare come l'applicazione risponde a tale evento.

La composizione del Vagrantfile è analoga al caso mono-macchina; le uniche differenze sono la presenza di più cicli di configurazione, uno per ciascuna macchina più uno generale, e la necessità di indicare quale delle

macchine sia quella primaria. Una volta create le macchine la loro gestione avviene con i comandi visti in precedenza che possono essere dati a tutte le macchine nel loro complesso oppure singolarmente. Sebbene la configurazione e l'utilizzo di un ambiente con più macchine virtuali di questo tipo risultino facili, tuttavia l'esecuzione di più macchine virtuali è pesante per un computer e vi è un numero limite di macchine virtuali che possono essere eseguite contemporaneamente, per questo motivo è necessario stabilire a priori quali servizi separare e quante macchine virtuali avere.

3.2 Docker

3.2.1 Overview

Docker è una piattaforma open source per lo sviluppo, la distribuzione e l'esecuzione di applicazioni che utilizza una tecnologia di virtualizzazione basata su container. Questo tipo di virtualizzazione utilizza il kernel del sistema operativo dell'host per eseguire più istanze guest. Ciascuna di queste istanze è chiamata container. Ogni container è un ambiente isolato che possiede un proprio filesystem, una certa quantità di memoria, dispositivi e configurazione di rete. In questo modo Docker fornisce l'abilità di impacchettare ed eseguire in maniera sicura ed isolata qualunque applicazione all'interno di un container. L'isolamento e la sicurezza permettono di poter eseguire più container simultaneamente su di uno stesso host. I container sono molto leggeri per quanto riguarda le risorse perchè essi non hanno bisogno del carico extra di un hypervisor, ma vengono eseguiti all'interno del kernel della macchina host. L'hypervisor è il componente, solitamente software, che si occupa di creare ed eseguire le macchine virtuali.[8] L'assenza di questo componente permette di eseguire più container su una certa

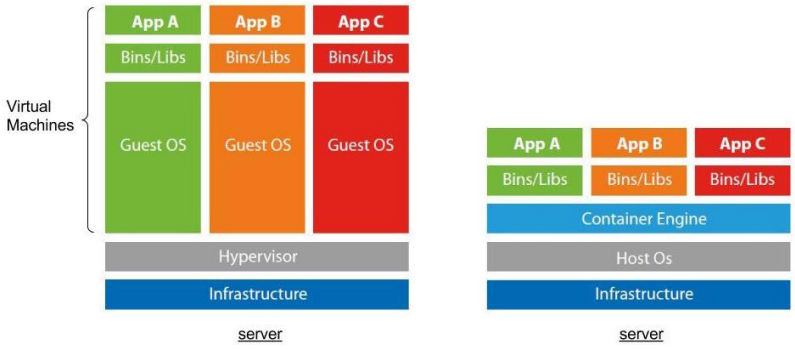

Figura 3.1: Differenza tra container e virtual machine

configurazione hardware rispetto all'utilizzo di macchine virtuali poiché rispetto a quest'ultime richiedono meno risorse in termini di CPU, memoria e spazio su disco. Inoltre è possibile eseguire i container di Docker proprio su macchine host che sono esse stesse macchine virtuali. Come mostrato in Figura 3.1, le macchine virtuali includono applicazioni, binaries, librerie e il sistema operativo necessario ad eseguire le applicazioni. In aggiunta a questo richiedono l'esecuzione di un hypervisor sul sistema operativo dell'host. Docker invece sfrutta il Docker Engine e i container che includono l'applicazione e tutte le loro dipendenze. I container condividono il kernel con gli altri container e vengono eseguiti come processi isolati nello spazio utente direttamente sul sistema operativo dell'host.

Una comparazione più dettagliata tra *virtual machine* e *linux container* si può trovare in [9]

3.2.2 Concetti e terminologia

Il nucleo della piattaforma Docker è composto dal *Docker Engine*. Il Docker Engine è una applicazione client-server che permette di creare e gestire

container. È composta da tre componenti principali:

- un server chiamato processo demone (*docker-daemon*), un programma che si occupa di creare e gestire oggetti Docker quali immagini, container, reti e volumi di dati sfruttando l'utilizzo delle caratteristiche del kernel di Linux.
- un'interfaccia REST API, per la comunicazione tra client e il demone
- il client che consiste in un programma esterno che prende gli input dell'utente e li invia al demone per creare, inviare o eseguire i container. In generale si tratta di una interfaccia di comando CLI ma può essere anche grafica [9].

Architettura

Docker utilizza una architettura client-server. Il client Docker parla con il demone il quale svolge il pesante lavoro di costruire, eseguire e distribuire i container[1]. Il client e il demone possono essere in esecuzione sullo stesso sistema, oppure è possibile connettere un client ad un demone remoto. Il client e il demone comunicano utilizzando REST API, attraverso socket UNIX o interfaccia di rete. Una rappresentazione grafica dell'architettura di Docker è mostrata in Figura 3.2.

Demone. Il demone (*docker-daemon*) rimane in ascolto per richieste API e gestisce gli oggetti di Docker come immagini, container, reti e volumi. In generale si ha un solo demone per container ma, come verrà spiegato a breve, si possono aver più demoni in esecuzione sullo stesso container. Un demone può anche comunicare con altri server per gestire i servizi di Docker.

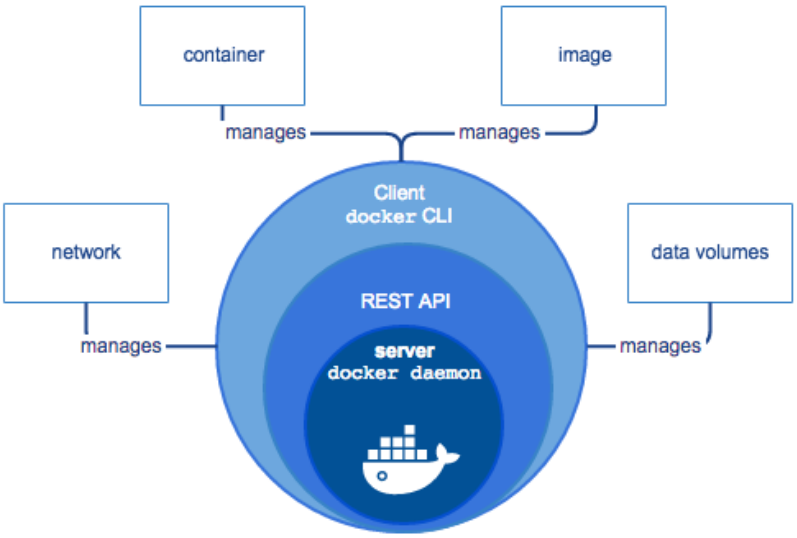

Figura 3.2: Architettura client-server del Docker Engine

Client. Il client di Docker è il metodo principale con cui più utenti Docker possono interagire con Docker. Utilizzando i comandi, come ad esempio `docker run`, il client invia i comandi al demone che li esegue. Il client accetta comandi dall'utente mediante una CLI e comunica con il demone Docker tramite REST API

Registers. Un registro Docker contiene le immagini Docker. Il Docker Hub[4] e il Docker Cloud[5] sono registri pubblici che chiunque può utilizzare e Docker è configurato per cercare immagini su Docker Hub di default. Inoltre è possibile utilizzare un proprio registro privato. Quando si utilizza il comando `docker pull` o `docker run`, le immagini richieste sono ottenute dal registro configurato. Per effettuare il *push* di una immagine

[4]https://hub.docker.com/
[5]https://cloud.docker.com/

al registro, si utilizza il comando `docker push`. Tramite Docker Store[6] è possibile acquistare e mettere in vendita immagini per Docker o distribuirle gratuitamente.

Objects

In questa sezione vediamo i principali elementi utilizzati sulla piattaforma Docker.

Immagini. Una immagine è un *template* in sola lettura che contiene le istruzioni per creare un *container* Docker. Spesso una immagine è basata su un'altra immagine con una personalizzazione addizionale. Ad esempio, è possibile costruire un'immagine basata su una immagine di Ubuntu, ma con installato un web server Apache. Lo sviluppatore può anche creare una propria immagine e renderla disponibile ad altri sviluppatori pubblicandola in un registro, come descritto in Sezione 3.2.2. Per costruire una immagine si deve creare un *Dockerfile*, come verrà descritto in Sezione 3.2.3. Proprio l'utilizzo dei Dockerfile rende le immagini più leggere e veloci, comparate ad altre tecnologie di virtualizzazione.

Containers. Un *container* è l'istanza eseguibile di una immagine. La differenza dal punto di vista tecnico tra un'immagine e un container è che quest'ultimo aggiunge un livello scrivibile in testa a tutti i livelli di sola lettura dell'immagine. Più container possono essere inizializzati da una stessa immagine. Tramite CLI o Docker API, è possibile creare un container, mandarlo in esecuzione, arrestare la sua esecuzione, spostarlo o eliminarlo. Si può collegare un container ad una o più reti, assegnargli spazio di memoria o creare una nuovo immagine basata sul suo stato

[6] https://store.docker.com/

corrente. Di default, un container è relativamente ben isolato dagli altri container e dalla sua macchina host. Lo sviluppatore può controllare come è isolata la rete del container, la memoria o gli altri sottosistemi ad esso legati dagli altri container e dalla macchina host. Un container viene definito al momento della sua creazione o esecuzione dalla sua immagine, così come ogni sua opzione di configurazione. Quando un container viene arrestato ogni cambiamento al suo stato che non viene salvato in memoria persistente scompare.

Services. I servizi permettono di scalare i container attraverso più demoni Docker, i quali lavorano tutti insieme come uno *swarm* con più *manager* e *workers*. Ogni membro di uno swarm è un demone Docker. I demoni comunicano tra di loro utilizzando le Docker API. Un servizio permette di definire uno stato desiderato come, ad esempio, il numero di repliche di un servizio che devono essere disponibili ad ogni dato istante. Di default, viene effettuato un *load-balancing* del servizio attraverso tutti i nodi *worker*. All'utente finale, il servizio Docker appare come una singola applicazione.

3.2.3 Dockerfile

Il Dockerfile è un file di configurazione che contiene le istruzioni per creare una immagine Docker. Tramite il Dockerfile l'utente scrive un insieme di istruzioni in un file, ad esempio installa un programma, monta un volume, e l'immagine viene costruita leggendo quel file. Se si vogliono utilizzare strumenti di orchestrazione per costruire un ambiente automatizzato, allora è necessario utilizzare un Dockerfile per la costruzione delle immagini. Un esempio molto semplice di un Dockerfile con solo tre istruzioni è mostrato qui sotto.

```
1 FROM ubuntu:16.04
2 RUN apt-get install curl
3 CMD ping 127.0.0.1
```

Vediamo di seguito che cosa vogliono dire nel dettaglio i comandi scritti nel file.

FROM. L'istruzione FROM specifica qual è l'immagine di base che viene utilizzata. In questo caso l'immagine di base è Ubuntu 16.04.

RUN. L'istruzione RUN è utilizzata per eseguire qualunque tipo di comando all'interno del container. Nell'esempio viene eseguito il comando per installare `cURL`[7]. Ogni istruzione che viene eseguita esegue il comando in un nuovo livello in cima al livello scrivibile e salva i cambiamenti in una nuova immagine.

CMD. L'istruzione CMD definisce il comando di default che deve essere eseguito ogni volta che un container viene inizializzato dall'immagine creata con quel Dockerfile. L'istruzione non ha nessuna influenza durante la fase di creazione ma l'istruzione cmd viene eseguita solo all'avvio del container. Per ogni Dockerfile può essere specificato solamente un comando cmd. Il comando può essere sovrascritto a run time quando il container viene avviato. Nell'esempio mostrato sopra, l'istruzione specifica che un comando `ping` viene eseguito quando il container viene avviato.

A partire dal Dockerfile un utente può creare una immagine tramite il comando `docker build`:

```
$ docker build [options] path
```

[7] https://curl.haxx.se/

Questo comando prende in input il *path* del *build context*. Il *build context* è la cartella che viene utilizzata come riferimento, ovvero la cartella in cui sono contenuti tutti gli elementi necessari per la fase di creazione. Durante la creazione il client Docker impacchetta tutti i file che sono salvati nel build context in un archivio .tar e lo invia al demone. Di default il demone ricerca il file chiamato Dockerfile nella root del build context e inizia a creare l'immagine leggendo le istruzioni.

Per gestire il ciclo di vita dei container e delle immagini Docker mette a disposizione diversi comandi, vediamo di seguito principali. Il comando docker start viene utilizzato per avviare un container che è già stato creato:

$ docker start <container ID>

Il comando `docker stop` arresta l'esecuzione di un container attivo:

$ docker stop <container ID>

Il comando `docker exec` esegue un comando all'interno di un container in esecuzione:

$ docker **exec** <container ID> <**command**>

Un container che non è in esecuzione può essere eliminato con il comando `docker rm` specificando l'ID o il nome del container:

$ docker rm [options] container [container ..]

Una immagine locale può essere eliminata con `docker rmi` specificando l'identificatore dell'immagine o il nome della repository:

$ docker rmi [options] image [image ...]

Una lista completa e dettagliata dei comandi si può trovare in [3].

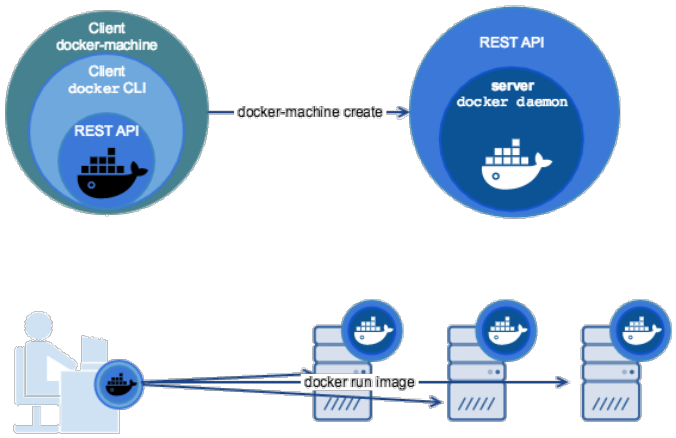

Figura 3.3: Relazione tra Docker machine e Docker engine

3.2.4 Orchestrazione: Docker Machine, Swarm e Compose

Con il termine orchestrazione ci si riferisce alla gestione, coordinazione ed organizzazione automatizzata di sistemi e servizi. Questo tipo di gestione nasce con dall'approccio di architettura orientata ai servizi (SOA), in cui i singoli servizi sono accoppiati in modo lasco e possono essere sviluppati con tecnologie diverse. Tali servizi infatti possono interagire anche con altri servizi non necessariamente legati alla stessa applicazione. L'orchestrazione è quindi fondamentale per coordinare tra loro tutti i diversi servizi e permettere a questi di comunicare tra di loro, cosa per la quale non sono programmati. Le versioni più recenti di Docker hanno portato ad aggiungere alla piattaforma alcuni strumenti per l'orchestrazione che vengono qui di seguito analizzati.

Docker Machine

La Docker Machine è uno strumento che permette all'utente di installare il Docker Engine su host virtuali e gestire gli host tramite comandi dedicati. Principalmente questo tool è utilizzato per creare ambienti Docker su macchine su cui non è presente Linux, quindi su sistemi operativi Windows e Mac OS, su *data center* o su *cloud provider* come Azure o Digital Ocean. Utilizzando i comandi `docker-machine` è possibile avviare, ispezionare, arrestare e riavviare un host, aggiornare il client e il demone e configurare il client per farlo comunicare con il proprio host. Facendo puntare la CLI della macchina ad un host in esecuzione è possibile eseguire comandi Docker direttamente su quell'host. A partire dalla versione 1.12 di Docker, le applicazioni Docker sono disponibili nativamente sia per Windows che per Mac OS, tuttavia per vecchi sistemi è necessario l'utilizzo di Docker Machine. In Figura 3.3 viene mostrato come la Docker Machine si interfaccia con il Docker Engine.

Docker Compose

Docker Compose è un *tool* per definire ed eseguire applicazioni composte da più container. Questo strumento risulta particolarmente efficace nel caso in cui si vuole realizzare una architettura basata su microservizi, dove ogni servizio necessario all'applicazione viene eseguito su un container dedicato e ciascun container viene collegato all'altro. Ogni servizio che andrà a comporre l'applicazione viene specificato in un file di configurazione YAML (*docker-compose.yml*). In questo file ogni servizio utilizzato nell'applicazione viene descritto da una serie di istruzioni che vengono utilizzate per realizzare ed eseguire il container che andrà a contenere quel partico-

lare servizio. Un esempio di file di configurazione per Docker compose è il seguente :

```
1  version: '2'
2  services:
3    web:
4      build: .
5      ports:
6       - "5000:5000"
7      volumes:
8       - .:/code
9    redis:
10     image: "redis:alpine"
```

Questo file docker-compose specifica due servizi: *web* (linea 3) e *redis* (linea 9). Il servizio web è composto da tre istruzioni: *build*, *ports* e *volumes*. Il servizio redis è invece composto solamente dalla istruzione *image*. Il comando `build` (linea 4) prende in input il path del Dockerfile che deve essere usato per realizzare l'immagine che ho deciso di usare per il container associato a quel servizio. Invece il comando `image` (linea 10), utilizzato per il servizio redis, prende in ingresso una immagine già esistente. Il comando `ports` (linea 5) specifica le porte che vengono esposte per il networking. Infine il comando `volumes` (linea 7) indica il percorso alla cartella designata ad essere utilizzata come data volume. Una volta realizzato il file di configurazione, `docker-compose` è possibile mandare in esecuzione l'applicazione tramite il seguente comando:

```
docker-compose up
```

Questo comando inizializza le immagini di ciascun servizio, crea i container e li avvia mettendo in connessione i servizi sulla stessa rete. Inoltre Docker compose permette di gestire l'intera infrastruttura in esecuzione mettendola in pausa, arrestando i container che compongono l'applicazione o riavviando i servizi. Tramite il comando pause è possibile mettere in pausa un particolare servizio:

```
docker-compose pause <nome servizio>
```

Docker Swarm

Swarm è una modalità di utilizzo del Docker Engine che è stata implementata nella versione 1.12.0. Questa modalità permette di gestire un cluster di Docker Engine, che viene chiamato appunto *swarm* (sciame). I Docker Engine che costituiscono lo swarm vengono chiamati nodi e ad essi vengono distribuiti i servizi. Tramite la CLI e le API di Docker è possibile gestire i nodi dello swarm, aumentandone o diminuendone il numero, e distribuire e orchestrare servizi. In modalità swarm si hanno due tipi di nodi: i nodi *manager* e i nodi *worker*. Almeno un nodo worker è sempre presente nello swarm e si occupa di inviare unità di lavoro (*task*) ai nodi worker. Inoltre il nodo manager si occupa di eseguire l'orchestrazione e la gestione delle funzioni del cluster per mantenere lo stato desiderato nello swarm. I nodi manager possono di default eseguire anche il ruolo di nodo worker, ma all'occorrenza possono essere configurati per eseguire solo il compito di manager. I nodi worker ricevono ed eseguono i task ricevuti dai nodi manager. Su questi nodi è in esecuzione un agente che effettua report sui task assegnati al nodo. Il nodo worker deve comunicare al nodo manager lo stato del task che gli è stato assegnato in modo tale che il nodo manager

possa mantenere lo stato desiderato di ogni nodo worker. In questa modalità i servizi assumono quindi il ruolo di task che vengono eseguiti sui nodi worker e rappresentano la struttura centrale del sistema swarm. I servizi possono essere replicati, in questo caso il manager dello swarm distribuisce uno specifico numero di repliche del task tra i nodi worker in base ad un fattore di scalamento desiderato, oppure globali, un task del servizio viene eseguito su ogni nodo disponibile del cluster.

Capitolo 4

Realizzazione servizi sandboxati

In questo capitolo vediamo come realizzare un ambiente sandboxato per una applicazione di *web security training*. Di seguito vengono presentate due possibili soluzioni basate sull'utilizzo dei due strumenti di virtualizzazione presentati nel Capitolo 3. Per entrambe verranno analizzate le scelte implementative e le differenze prestazionali.

4.1 Use case: Cyber-gym

Per ovviare alle vulnerabilità di *input validation* esistono diverse guide che descrivono test specifici da eseguire sulla propria applicazione. Tra queste guide la più famosa è la Testing Guide di OWASP [1]. Sebbene questi test siano molto affidabili e completi, la loro esecuzione risulta dispendiosa in termini di tempo e poco efficiente, soprattutto se consideriamo il fatto che un eventuale fallimento nel test comporterebbe il ritorno alla fase di

[1] https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents

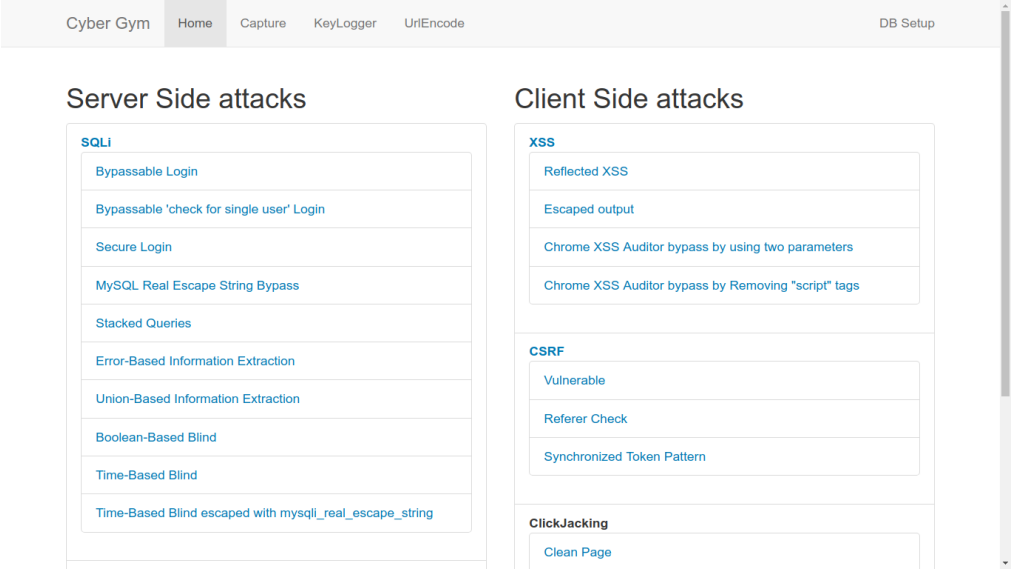

Figura 4.1: Home Page dell'applicazione Cyber-gym

sviluppo. Le capacità e la conoscenza dell'argomento da parte degli sviluppatori può quindi fare la differenza in fase di sviluppo e velocizzare la realizzazione dell'applicazione. Per aiutare gli sviluppatori e chi si occupa di web application e sicurezza, ma anche come metodo di insegnamento e di esercizio per gli studenti, sono state sviluppate diverse applicazioni web deliberatamente vulnerabili. Queste applicazioni permettono infatti di esercitarsi e provare ad effettuare in modo legale attacchi sfruttando le diverse vulnerabilità di input validation in un ambiente costruito ad hoc. Tra le applicazioni più famose che si hanno a disposizione abbiamo Mutillidae[2], sviluppata da OWASP, e la DVWA[3] (Damn Vulnerable Web Application).

L'applicazione che però prendiamo in considerazione è Cyber-gym[4].

[2]https://sourceforge.net/projects/mutillidae/
[3]http://www.dvwa.co.uk/
[4]https://github.com/AvalZ/Cyber-gym

Cyber-gym è una web application realizzata in PHP che contiene diversi scenari/test, ciascuno dei quali è dedicato ad una particolare vulnerabilità e si presta ad un determinato tipo di attacco. In particolare questa applicazione si concentra su vulnerabilità di SQL Injection e XSS. In Figura 4.1 è mostrata la home page dell'applicazione dove sono elencati i diversi scenari/test che un utente può eseguire. Cyber-gym è una applicazione prevalentemente didattica, infatti è possibile andare ad analizzare gli script in PHP di ciascuna pagina o scenario ed individuare le vulnerabilità a partire dall'analisi del codice. All'occorrenza un utente può a suo piacimento andare ad apportare modifiche al codice stesso per eliminare una certa vulnerabilità e quindi testare le proprie capacità e conoscenze. Un ulteriore vantaggio di questa applicazione è il fatto di essere realizzata all'interno di un ambiente virtuale in Vagrant già preimpostato che rende l'avvio e l'utilizzo dell'applicazione molto semplice ed immediato a differenza delle altre applicazioni citate in precedenza che lasciano all'utente l'onere di crearsi un ambiente virtuale per essere eseguite.

4.2 Panoramica

Sfruttando la natura esercitativa dell'applicazione Cyber-gym, si è deciso di modificarne l'architettura in modo da realizzare un sistema sandboxato automatizzato. Il sistema deve riconoscere se un utente ha eseguito in modo corretto un attacco di SQL Injection ed in tal caso, dopo averne mostrato i risultati, procede con il ripristino dell'ambiente di esercitazione. Basandosi sull'architettura originale dell'applicazione, il metodo più semplice ed immediato per realizzare questo sistema di "recovery" è quello di avere più duplicati dell'applicazione pronti per essere utilizzati. L'utente accede ad

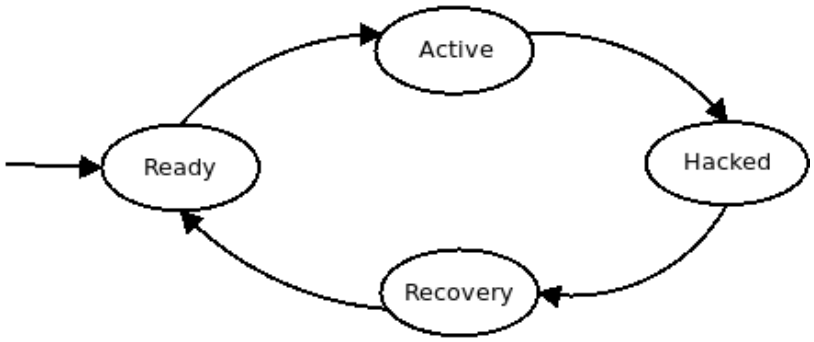

Figura 4.2: Ciclo di vita delle repliche

una delle copie dell'applicazione ed effettua la prova di SQL Injection; nel caso in cui questa vada a buon fine la copia utilizzata dall'utente verrà ripristinata. Tuttavia l'operazione di ripristino non è immediata e richiede tempo, per cui l'utente viene reindirizzato su una delle altre copie disponibili che si trovano nel loro stato originale. In questo modo l'utente non deve attendere nessun tempo di ripristino e può effettuare un nuovo test mentre in background viene ripristinata la copia appena utilizzata.

In Figura 4.2 possiamo vedere il ciclo di vita di ciascuna copia dell'applicazione e gli stati in cui essa si può trovare. Lo stato *ready* rappresenta lo stato iniziale in cui si trova l'applicazione all'avvio del sistema o dopo un eventuale ripristino; quando una copia si trova in questo stato è pronta per essere utilizzata e allora passa allo stato *active*. La copia dell'applicazione permane in questo stato finché l'utente non effettua con successo una SQL Injection A questo punto si entra nello stato *hacked*, in cui l'applicazione ha subito modifiche non autorizzate; in questo stato l'utente può verificare quali siano stati gli effetti dell'attacco sull'applicazione. Una volta che la copia si trova nello stato *hacked*, non può più essere utilizzata per effettuare altri attacchi e deve quindi essere riportata allo stato originale per essere

nuovamente disponibile. Pertanto la copia passerà dallo stato *hacked* allo stato *recovery* nel quale si effettueranno tutte le necessarie operazioni di ripristino. Una volta terminate queste operazioni, la copia dell'applicazione tornerà nello stato *ready* pronta per essere utilizzata nuovamente.

4.3 Realizzazione con Vagrant

In questa sezione viene mostrata una prima realizzazione dell'ambiente del sistema appena descritto. Questa soluzione si basa sul tool di virtualizzazione Vagrant.

4.3.1 Struttura della soluzione

Come mostrato nel Capitolo 3.1, Vagrant offre la possibilità di realizzare più macchine virtuali collegate in rete tra loro, utilizzando un solo Vagrantfile. Con dei semplici comandi da terminale, è possibile gestire l'intero ciclo di vita delle singole macchine. Sfruttando queste caratteristiche dell'ambiente di virtualizzazione, è possibile realizzare il sistema di *recovery* in modo che ciascuna copia dell'applicazione risieda su una macchina virtuale separata. Ogni copia dell'applicazione corrisponderà ad una macchina virtuale isolata rispetto alle altre macchine e l'indirizzamento dell'utente a ciascuna di esse sarà mediato da un server *proxy*.

Un server *proxy* svolge la funzione di intermediario tra le richieste dei client che ricercano risorse da altri server. Un client si connette al server *proxy* richiedendo un qualche tipo di servizio, ad esempio una pagina web o un file disponibile su un altro server. Quest'ultimo valuta ed esegue la richiesta in modo da semplificare e gestire la a complessità. Un server proxy può risiedere sul pc locale dell'utente o in diversi punti tra il client

e il server di destinazione sulla rete. In generale si distinguono 3 tipologie di server proxy:

Gateway un server *proxy* che passa richieste e risposte senza modificarle.

Forward proxy un proxy rivolto all'utilizzo su Internet e al recupero di dati da un vasto campo di risorse.

Reverse proxy è rivolto all'uso su internet ed è usato in front-end per controllare e proteggere gli accessi ai server su una rete privata

Il proxy server realizzato per l'applicazione Cyber-gym ha il compito di indirizzare le richieste dell'utente alla macchina corretta, ovvero ad una di quelle che contengono una copia dell'applicazione nello stato originale e che risulta disponibile al momento della richiesta. Inoltre lo stesso proxy si occupa di gestire il ciclo di vita delle macchine del sistema, infatti alla ricezione di un segnale opportuno il proxy provvede al riavvio delle macchine.

4.3.2 Implementazione del server proxy

Il server proxy è stato realizzato in Node.js, una piattaforma event-driven che consente di eseguire codice Javascript lato server di cui si è discusso in Sezione 1.3.1. Di seguito viene mostrato il codice che implementa il server proxy.

```
1 var vagrant = require('vagrant');
2 var httpProxy = require('http-proxy');
3 var http = require('http');
4 var express = require('express');
5 var app = express();
```

```
6  var jsonObj = require("./Cyber-gym/Machines.json");
7  var iterator = 1;
8
9  vagrant.up(function(code) {
10     console.log('Cyber-gym is ready');
11 });
12
13 app.get('/', function (req, res) {
14 var target_address = getTargetAddress();
15   proxy.web(req, res, { target: target_address });
16 });
17
18 app.get('/get_status', function(req, res){
19   vagrant.status(function(code){
20   });
21   res.end();
22 });
23
24 app.get('/restart_machine',function(req,res){
25   var oldID = iterator;
26   var oldAddress = jsonObj['Address'+iterator];
27   var oldName = jsonObj['Machine'+iterator];
28
29   delete jsonObj['Address'+iterator];
30   delete jsonObj['Machine'+iterator];
31   if (iterator == 4 ) {
32     iterator = 1;
```

```
33    }
34    else {
35      iterator++;
36    }
37  res.end();
38  restartMachine(oldID, oldAddress, oldName);
39  });
40  //
41  ...
42  //
43  var proxy = httpProxy.createProxyServer({});
44  var server = http.createServer(app, function(req, res
        ) {
45  });
46  server.listen(8080);
```

Nelle linee da 1 a 5 vengono dichiarate le variabili per i moduli utilizzati. Una funzionalità molto importante di Node.js è infatti quella di poter utilizzare dei moduli preconfigurati che vengono installati tramite semplici comandi da terminale. Un modulo può essere paragonato ad una libreria di Javascript, ovvero è un insieme di funzioni che si vogliono includere nella propria applicazione. Per il proxy server in questione sono stati utilizzati i seguenti moduli:

- express,
- http,
- http-proxy,

- vagrant.

Express.js è un framework per le applicazioni web che fornisce un grande insieme di funzionalità per facilitare e velocizzare lo sviluppo di applicazioni web e mobile. Questo modulo infatti permette di impostare dei middleware per rispondere a richieste HTTP, definire una *routing table* utilizzata per effettuare diverse azioni basate su metodi HTTP e URL, permette di renderizzare dinamicamente pagine HTML. Il modulo *http* include una serie di oggetti utili per sfruttare al meglio il protocollo HTTP e serve soprattutto per creare e gestire server. Analogamente il modulo *http-proxy* è una libreria utile per implementare *proxy* e *load balancer*. Infine, il modulo *vagrant* è una libreria che permette di utilizzare funzioni "wrapper" per la command line di Vagrant, ovvero permette di richiamare comandi da terminale direttamente dal server. Il sistema viene avviato eseguendo su terminale il comando `node server.js` il quale manda in esecuzione il file sopra riportato. In questo modo viene inizializzato sia il server proxy sia le varie macchine su cui si trovano le copie dell'applicazione.

Alla linee 44 infatti, troviamo la parte di codice che ci permette di creare un server HTTP in ascolto sulla porta 8080. Questo viene fatto dichiarando una variabile server e chiamando la funzione *createServer*, la quale crea appunto un oggetto web server. La funzione che è passata all'interno di questo metodo viene chiamata ogni volta che viene effettuata una richiesta a quel server e per questo viene chiamata request handler. Infatti l'oggetto server ritornato da *createServer* è un *event emitter*. Ogni volta che una richiesta http arriva al server la funzione di *request handler* gestisce la transazione. Per servire la richiesta è necessario chiamare il metodo *listen* (linea 45) per l'oggetto server indicando il numero della porta sulla quale il server rimane in ascolto.

Alla linea 43 viene creato il proxy utilizzando la funzione *createProxyServer* del modulo *http-proxy*. Le richieste al proxy vengono gestite tramite il metodo web (linea 15). Questo metodo è presente all'interno della funzione che si attiva alla richiesta della pagina iniziale (linee 13-16). Node.js utilizza il concetto di *routing* per determinare come l'applicazione debba rispondere ad una determinata richiesta da parte del client verso un particolare *endpoint* il quale è costituito da un certo URI e da un metodo HTTP (GET, POST). In questo caso la richiesta è di tipo GET e l'URI è '/' che indica la pagina *index*. Utilizzando il metodo che invoca il proxy la pagina che viene restituita è quella del server corrispondente all'indirizzo a cui punta il target. Il valore del *target* è l'indirizzo corrispondente alla macchina sulla quale si trova una copia dell'applicazione. Ovviamente la macchina dovrà essere una di quelle che si trovano in uno stato ready e quindi il target deve essere scelto in maniera opportuna. Per questo motivo l'indirizzo target viene scelto da un elenco di indirizzi corrispondenti a macchine esistenti e che si trovano nello stato di disponibilità. Questo elenco è contenuto in un opportuno file JSON (linea 6). All'avvio del sistema tutte le macchine saranno ovviamente disponibili e pertanto i loro indirizzi saranno presenti nell'elenco. Quando una delle macchine deve essere riavviata l'indirizzo corrispondente viene eliminato temporaneamente dall'elenco. Quando la macchina tornerà disponibile questa invierà una richiesta al server proxy e il suo indirizzo verrà nuovamente inserito nell'elenco delle macchine disponibili. La gestione di questo elenco così come il riavvio delle macchine viene gestito dal proxy server. Sfruttando il concetto di routing è stata appunto realizzata una funzione che gestisce la richiesta di rigenerazione di una macchina (linee 24-39). L'applicazione (o per meglio dire la copia dell'applicazione) attiva in un certo istante, dopo che

l'utente ha eseguito correttamente un attacco di SQL Injection, invierà una richiesta http al proxy con l'obiettivo di essere riavviata e riportata allo stato iniziale. Questa richiesta viene gestita dal proxy, il quale ottenendo l'indirizzo IP della macchina che ha effettuato la richiesta esegue prima il comando di `vagrant destroy` per eliminare la macchina e successivamente il comando di `vagrant up` per renderla nuovamente disponibile per essere utilizzata. Queste operazioni vengono eseguite in maniera asincrona. Durante questo procedimento viene opportunamente modificata la lista degli indirizzi. Prima di eseguire il comando di destroy l'indirizzo della macchina viene eliminato dall'elenco di quelle disponibili(linee 29-30) e una volta tornata disponibile la macchina il suo indirizzo viene nuovamente inserito nell'elenco.

Le varie macchine Vagrant vengono inizializzate all'avvio insieme al proxy server sfruttando il modulo Vagrant per Node.js. All'esecuzione del comando di avvio del proxy infatti viene eseguito il metodo alla linea 9 che corrisponde ad eseguire su terminale il comando `vagrant up`, quindi inizializza tutte le macchine presenti sul Vagrantfile che gli viene passato come argomento. Quando tutte le macchine sono avviate l'utente viene avvisato tramite un messaggio sul terminale (liea 10).

Infine è stata aggiunta la possibilità di controllare lo stato di tutte le macchine su richiesta dell'utente (linee 18-22) tramite il "wrapping" del comando `vagrant status`.

Il codice completo è disponibile in Appendice.

4.3.3 Caso d'uso: attacco stacked queries

Tutto queste operazioni vengono eseguite lato server e ovviamente i metodi GET e POST vengono eseguiti in seguito a richieste HTTP provenienti

dal client. In questo caso le richieste vengono inviate dall'applicazione stessa. Allo script originale mostrato in dettaglio nel Capitolo 2.3.2. è stato aggiunta la seguente parte di codice:

```
1  $query_check_end = 'CHECKSUM TABLE accounts';
2   if($con->query($query_check_end)){
3      $result = $con->query($query_check_end);
4      $row = $result->fetch_assoc();
5      $endChecksum = $row["Checksum"];
6      if($endChecksum==NULL){
7         echo "OK Checksum = NULL <br>";
8      }
9      else{
10        echo"***error on retrieving checksum*** <br>";
11     }
12  if($initChecksum != $endChecksum){
13     echo "Database modificato <br>";
14     echo "<form action='reload.php'>";
15     echo "<input type='submit' value ='Reload'>";
16     echo "</form>";
17  }
```

Questa parte permette di controllare la corretta riuscita dell'attacco da parte dell'utente viene controllata dallo script tramite un confronto tra il checksum del database ottenuto prima dell'esecuzione della query e checksum ottenuto dopo l'esecuzione della query (linea 12). Il checksum viene solitamente utilizzato prima e dopo l'esecuzione di backup per controllare che l'operazione sia stata eseguita in modo corretto. Questa istruzione è utilizzata sporadicamente in quanto, a seconda di cosa gli ordiniamo di

controllare, va a leggere ogni elemento di ciascuna tabella e, come è facile intuire, più è grande il database più tempo viene impiegato per portare a termine l'operazione. Fortunatamente il database dell'applicazione Cyber-gym ha dimensioni ridotte ed il tempo impiegato per effettuare le due operazioni non è eccessivo. Se viene riscontrata una differenza tra i due checksum allora l'attacco è stato eseguito correttamente.

Per come è stata definito in precedenza il comportamento del sistema, l'applicazione a questo punto mostrerà un messaggio all'utente indicando appunto la riuscita dell'attacco e mostrando la query che l'utente ha generato per l'attacco. A seguito della modifica del database la copia dell'applicazione utilizzata a questo punto deve essere riportata allo stato originale. Quello che il sistema deve fare a dopo aver rilevato la modifica al database è inviare una richiesta AJAX al server proxy per eliminare dalla lista delle macchine utilizzabili l'indirizzo IP della macchina corrente. La macchina non viene riavviata autenticamente per permettere di mostrare la pagina dei risultati all'utente, tuttavia viene consigliato all'utente di tornare alla home page utilizzando un apposito bottone(linea 14) che permette l'invio della richiesta AJAX al server proxy per eseguire il riavvio della macchina.

L'utente viene quindi reindirizzato alla home page dell'applicazione ma questa sarà una copia diversa da quella appena utilizzata.

4.3.4 Considerazioni sulla soluzione

L'approccio che è stato utilizzato nel realizzare la soluzione appena analizzata è stato quello di mantenere come base il sistema di virtualizzazione di Vagrant. L'applicazione Cyber-gym infatti nasce come un ambiente isolato in cui poter esercitare le proprie conoscenze di web security in quello che è di fatto è un laboratorio virtuale costruito su misura per questo scopo.

Come discusso nel Capitolo 3.1, Vagrant nasce come strumento per realizzare ambienti sandbox isolati e virtuali su misura per l'utente. Tuttavia dovendo orchestrare e gestire il ciclo di vita delle macchine virtuali realizzare un sistema racchiuso completamente in un ambiente virtualizzato non è la soluzione migliore ed inoltre è difficilmente percorribile. I comandi da terminale per gestire le macchine Vagrant infatti devono necessariamente essere eseguiti sulla macchina fisica sulla quale è in esecuzione l'ambiente di virtualizzazione. Il motivo principale è che ovviamente è necessario che il software di Vagrant sia installato correttamente, inoltre avrebbe poco senso il poter utilizzare alcuni comandi, come ad esempio quello per la distruzione della macchina virtuale, dall'interno della macchina virtuale stessa. Tuttavia dovendo gestire un insieme di macchine virtuali potrebbe tornare utile gestire una macchina virtuale tramite un'altra macchina virtuale. Ad esempio, nel sistema di recovery realizzato, una possibile variante potrebbe essere quella di inserire il proxy server all'interno di un ambiente virtuale Vagrant. In questo caso i comandi per la gestione della macchina virtuale dovrebbero essere scambiati tra una macchina virtuale e l'altra. Questo scambio di comandi differisce dal normale utilizzo di Vagrant e non è facilmente implementabile. Durante la fase di sviluppo del sistema di backup è stato trovato un metodo per ovviare a questo problema che consiste nell'andare a sfruttare il provisioning, le cartelle condivise e la connessione ssh che si crea tra la macchina virtuale e la macchina fisica. Questo metodo non è stato però utilizzato in quanto l'utilizzo di una ulteriore macchina virtuale solo per il server proxy è superfluo e non è consigliabile abusare nell'utilizzo di macchine virtuali per evitare di sovraccaricare la macchina dell'utente. La soluzione realizzata con Vagrant infatti ha il difetto di dover avere molte macchine virtuali costantemente in esecuzione in quanto

il ripristino di una di esse non avviene in modo immediato ma necessita di alcuni minuti in base alle prestazioni della macchina fisica che ospita il sistema. Sebbene le macchine virtuali utilizzate non abbiano interfaccia grafica (headless) incidono pesantemente sulle prestazioni del sistema su cui vanno in esecuzione ed è necessario limitarne il numero.

Per questo motivo si è deciso di proporre una seconda soluzione, sempre basata sulla virtualizzazione ma che risultasse più efficiente sia dal punto di vista delle prestazioni sia dal punto di vista del carico computazionale sul sistema.

Nella Sezione 4.4 viene descritta l'implementazione in Docker.

4.4 Realizzazione con Docker

4.4.1 Introduzione

La seconda soluzione proposta per la realizzazione del sistema di recovery è basata sulla piattaforma Docker. Come discusso nel capitolo 3.2 Docker è un piattaforma di virtualizzazione che si basa sull'utilizzo di container. I container sono ambienti isolati paragonabili a macchine virtuali, ma a differenza di queste non necessitano di un *hypervisor* in esecuzione sul sistema operativo dell'host e questa caratteristica rende i container molto meno avidi di risorse rispetto alle classiche macchine virtuali. Un'ulteriore caratteristica molto interessante di Docker è la possibilità di utilizzare la modalità Swarm. Questa modalità permette di gestire in modo semplice un cluster di Docker container che in questo caso assumono il ruolo di nodi. La modalità Swarm di Docker inoltre permette di effettuare l'orchestrazione dei servizi e il load balancing in modo semplificato andando a gestire automaticamente le repliche dei servizi che l'utente manda in ese-

cuzione. Queste caratteristiche della modalità Swarm di Docker possono essere sfruttate per andare a rendere più efficiente il meccanismo di gestione dell'applicazione Cyber-gym. Il concetto di base rimane invariato; come per la prima soluzione si avrà un insieme di copie dell'applicazione Cyber-gym alle quali si accederà in modo selettivo tramite l'utilizzo di un proxy, ma in questa soluzione le copie dell'applicazione non saranno più rappresentate da una macchina virtuale Vagrant ma saranno servizi in un Docker Swarm.

4.4.2 Realizzazione componenti Docker

Per la realizzazione di questa soluzione si è scelto di realizzare uno Swarm composto da un solo nodo worker sul quale verranno eseguiti tutti i servizi, inoltre questo nodo viene realizzato tramite una macchina virtuale Vagrant. Questa scelta è stata fatta per mantenere il sistema il più semplice possibile e per sfruttare al meglio non solo le caratteristiche di Docker ma anche quelle di Vagrant. La macchina virtuale sulla quale andrà in esecuzione il nodo dello Swarm viene realizzata tramite il Vagrantfile seguente:

```
1 Vagrant.configure("2") do |config|
2
3     config.vm.box = "scotch/box"
4
5 config.vm.define "original", primary: true do |
    original|
6     original.vm.network "private_network", ip: "
        192.168.33.10"
7     original.vm.hostname = "scotchbox"
```

```
8       original.vm.synced_folder ".", "/var/www",
            :mount_options => ["dmode=777", "fmode=666"]
9       original.vm.provision: "shell", path: "docker-
            install.sh"
10      end
11  end
```

Questo Vagrantfile è simile a quello mostrato in 4.3: viene realizzata una macchina virtuale chiamato original a cui viene assegnato l'indirizzo IP: 192.168.33.10 e il modello utilizzato è quello di scotchbox. La differenza sostanziale con le macchine del modello precedente è nello script docker-install.sh eseguito nel provisioning(linea 9). Questo script procede all'installazione all'interno della macchina virtuale di tutte le componenti di Docker necessarie per utilizzare la modalità Swarm: per prima cosa si procede all'installazione del Docker Engine e successivamente all'installazione del tool Docker Compose.

In Docker per andare a definire i servizi da mandare in esecuzione su un nodo si utilizza il file Dockercompose di cui si è discusso in Sezione3.2.4. Tramite il seguente file vengono quindi definite le caratteristiche delle diverse copie dell'applicazione Cyber-gym che saranno in esecuzione sul nostro sistema.

```
1   web1:
2       image:nginx
3       volumes:
4         - .:/var/www
5       ports:
6           - "8000:8000"
```

```
7
8  web2:
9      image: nginx
10     volumes:
11       - .:/var/www
12     ports:
13         - "8001:8000"
14
15 web3:
16     image: nginx
17     volumes:
18       - .:/var/www
19     ports:
20         - "8002:8000"
```

Questo file dockercompose definisce tre servizi chiamati rispettivamente web1, web2, web3 i quali rappresentano tre copie dell'applicazione Cybergym. Per tutti e tre i servizi viene definita l'immagine docker (linea 1) da utilizzare e dove si trova il codice della nostra applicazione (linea 2). Infine a ciascun servizio viene indicata una porta sulla quale rimanere in ascolto che ovviamente è diversa per ogni servizio.

Ultimo elemento necessario per la realizzazione del sistema è il proxy server, anche in questo caso realizzato in Node.js. Di seguito sono riportate le parti di codice principali, il codice completo è riportato in AppendiceA.1.

```
1 var vagrant = require('vagrant');
2 var httpProxy = require('http-proxy');
3 var http = require('http');
```

```
4  var express = require('express');
5  var app = express();
6  var jsonObj = require("./Cyber-gym/Services.json");
7  var iterator = 1;
8
9  vagrant.up(function(code) {
10     console.log('Cyber-gym is ready');
11 });
12
13 app.get('/', function (req, res) {
14 var target_address = getTargetAddress();
15   proxy.web(req, res, { target: target_address });
16
17 });
18
19 app.get('/restart_service',function(req,res){
20   var oldID = iterator;
21   var oldPort = jsonObj['Port'+iterator];
22   var oldName = jsonObj['Service'+iterator];
23
24   delete jsonObj['Port'+iterator];
25   delete jsonObj['Service'+iterator];
26   if (iterator == 6 ) {
27     iterator = 1;
28   }
29   else {
30     iterator++;
```

```
31    }
32  res.end();
33  restartService(oldID, oldPort, oldName);
34  });
35
36  function restartService(id, port, name){
37  vagrant.ssh('virtual -c sudo docker remove' + name,
        function(code){});
38
39  vagrant.ssh('virtual -c sudo docker-compose.yml' +
        name, function(code){});
40  }
41
42  var proxy = httpProxy.createProxyServer({});
43  var server = http.createServer(app, function(req, res
        ) {
44  });
45  server.listen(8085);
```

Questa versione del proxy utilizza gli stessi moduli visti nella versione precedente(4.3.2): express, http, http-proxy e Vagrant. Analogamente, alla linea 9 ritroviamo la funzione che genera il comando `vagrant up` per l'avvio della macchina virtuale e la funzione di creazione del proxyserver (linea 42). A differenza del caso precedente non sono più necessarie le funzioni che riavviano le macchine virtuali in Vagrant, ma sono presenti nuove funzioni che si occupano di gestire il ciclo di vita dei servizi all'interno del nodo dello swarm. Queste funzioni vengono chiamate come nel caso precedente tramite la funzionalità di routing del modulo express e tramite l'utilizzo

del modulo Vagrant. La funzione alla linea 36 viene richiamata per il riavvio del servizio attualmente in uso, essa invia tramite ssh il comando di rimozione del servizio alla macchina Vagrant (linea 37).Una volta rimosso il servizio questo viene automaticamente riavviato sempre tramite l'invio dell'apposito comando tramite ssh(linea 39). Come nel caso precedente è presente a lista in JSON delle copie dell'applicazione attive che viene aggiornata in modo automatico durante il processo(linea 6). In questo caso questo caso la lista non contiene più la corrispondenza nome macchina e indirizzo IP ma conterrà il nome del servizio e la porta sulla quale rimane in ascolto.

4.4.3 Caso d'uso: attacco Stacked Queries

Il comportamento in backend dell'applicazione quindi non è molto diverso rispetto alla prima soluzione. Utilizzando come caso d'uso l'attacco di stacked queries mostrato in sezione 4.3.3 possiamo evidenziare le principali differenze tra le due soluzioni. Il metodo utilizzato per rilevare un attacco eseguito in modo corretto rimane invariato, anche in questo caso si utilizza un confronto tra i checksum del database prima e dopo l'esecuzione della query. Se l'attacco va a buon fine il sistema provvede al riavvio tramite l'invio di una richiesta AJAX come nel caso precedente.

In questo caso il server proxy riceve la richiesta per il riavvio del servizio e procede eliminando il servizio dall'elenco di quelli disponibili e inviando l'opportuno comando per la rimozione del servizio dal nodo dello swarm. Una volta che la rimozione di quest'ultimo viene confermata si procede al riavvio di tale servizio. Per la realizzazione di questa soluzione sono state fatte alcune scelte implementative che è necessario approfondire. Per realizzare le repliche della applicazione si è scelto di andare a definire ciascuna

copia con un servizio diverso nel Dockerfile, ma come si è visto in Sezione 3.2.4 è possibile dichiarare nello stesso file dockercompose quante repliche di un servizio vogliamo mandare in esecuzione nello swarm. Inoltre durante l'esecuzione è possibile effettuare lo scaling di questi servizi. Tuttavia implementando i servizi in questo modo non è possibile gestire quest'ultimi in modo indipendente tra di loro e con la libertà necessaria ai nostri scopi. Questo perchè la modalità swarm di docker nasce fondalmentalmente per fare load balancing e garantire la continuità dei servizi. Il docker engine infatti in questa modalità garantisce che in ogni momento ogni servizio sia in esecuzione su uno o più nodi nel numero di repliche indicato sul file dockercompose, inoltre eventuali modifiche ad un servizio vengono riportate anche alle altre repliche. Risulta pertanto evidente come sia necessario dover ricorrere alla soluzione di elencare un servizio per ciascuna copia dell'applicazione Cyber-gym all'interno del dockerfile. La scelta di utilizzare Docker all'interno di una macchina virtuale Vagrant, ci permette in primis di semplificare l'implementazione dell'ambiente sandboxato, la quale viene automatizzata tramite le funzionalità di provisioning fornite da Vagrant già viste nella prima soluzione e non richiede all'utente di procedere autonomamente all'installazione dei diversi software e alla configurazione del sistema. Inoltre questa scelta ci permette di mettere in evidenza uno dei vantaggi che questa l'utilizzo di docker swarm ci da rispetto alla prima soluzione di Vagrant e che verranno discussi nel capitolo 4.5, ovvero la riduzione del numero di macchine virtuali in esecuzione sul sistema dell'utente. Infine lo swarm che viene realizzato è composto da un solo nodo: questa scelta si ricollega ad entrambe le considerazioni precedenti. Avendo scelto di mantenere una virtualizzazione con Vagrant l'aggiunta di un secondo nodo allo swarm comporterebbe la necessità di implementare una

seconda macchina virtuale e di conseguenza modificare il comportamento del proxyserver che dovrebbe gestire un indirizzamento per le macchine virtuali e un indirizzamento per i servizi. Inoltre l'introduzione di un secondo nodo nello swarm renderebbe necessaria la realizzazione di una politica di load balancing adeguata che con un solo nodo risulta superflua. Questo tipo di soluzione, come indicato nel Capitolo 5, può essere interessante da sviluppare in futuro.

4.5 Confronto tra le due soluzioni

Come abbiamo visto nelle sezioni precedenti, entrambe le soluzioni ci permettono di realizzare un ambiente sandboxato e automatizzato per l'applicazione di web security training Cyber-gym. Andando ad analizzare i due approcci dal punto di vista delle prestazioni e dell'utilizzo delle risorse troviamo conferma di quanto anticipato nella Sezione 3.2 per quanto riguarda le differenze nella virtualizzazione tra Vagrant e Docker.

CPU	Intel® Core™ i7-2630QM @ 2.0GHz x 8
Memoria RAM	8 GB
Memoria RAM libera	6,2 GB
Grafica	Intel® Sandybridge Mobile
S.O.	Ubuntu 17.10 64-bit
Disco	120 GB

Tabella 4.1: Specifiche del computer usato per i test

Nella Tabella 4.1 sono riportate le caratteristiche della macchina fisica utilizzata sia per lo sviluppo dell'ambiente di sandboxing sia per i test presentati in questa sessione.

Analizziamo per prima la soluzione realizzata con più macchine Vagrant. Nella 4.2 tabella troviamo i dati relativi alle risorse messe a disposizione di una singola macchina virtuale di Vagrant nella sua configurazione standard. Analizzando il comportamento della macchina a regime si è evidenziato uno spreco di tali risose, che possono essere ottimizzate tramite il ridimensionamento mostrato sempre nella Tabella 4.2. Da questo confronto possiamo notare come l'ottimizzazione riguardi un ridimensionamento sia della potenza di calcolo che della memoria virtuale messe a disposizione della macchina virtuale. La diminuzione delle risorse assegnate alla macchina virtuale non influenza in alcun modo il funzionamento della applicazione in esecuzione su di essa.

	Configurazione standard	Configurazione ottimizzata
S.O	Ubuntu 14.04.5 LTS 64-bit	Ubuntu 14.04.5 LTS 64-bit
CPU	2	1
Memoria RAM	2 GB	1 GB
Disco	40 GB	20 GB

Tabella 4.2: Configurazioni della macchina virtuale Vagrant

Procedendo con l'esecuzione di più copie dell'applicazione e quindi eseguendo più macchine virtuali si raggiunge una saturazione delle risorse della macchina fisica con 10 macchine virtuali attive. Inoltre è importante evidenziare che aumentando il numero di macchine virtuali aumenta anche il tempo per il deploying di tali macchine quando viene eseguito il comando di Vagrant up all'avvio dell'applicazione. Nella tabella 4.3 vengono riportati i valori relativi al consumo della memoria RAM della macchina fisica e il tempo impiegato per l'avvio completo del sistema al variare del numero di macchine virtuali. In modo analogo possiamo analizzare le prestazioni

N repliche attive	1	3	5	10
Percentuale utilizzo CPU	8%	28%	40%	80%
Memoria RAM utilizzata	0,7 GB	1,9 GB	2,9 GB	6,1 GB
Tempo di avvio	20-30 sec	70-100 sec	1 min	2 min

Tabella 4.3: Risorse utilizzate al variare del numero di macchine attive

per la seconda soluzione con delle piccole variazioni. A differenza del caso precedente, le risorse assegnate alla macchina virtuale vanno analizzate in modo tale da essere massimizzate e non minimizzate. Nel primo caso infatti è necessario ridurre le risorse consumate da ogni singola macchina per aumentare il numero di macchine attivabili contemporaneamente sulla macchina fisica. In questo caso infatti si ha una sola macchina virtuale che deve contenere al suo interno il maggior numero possibile di repliche dell'applicazione, quindi è necessario aumentare le risorse a disposizione della macchina virtuale. Per questo motivo le risorse assegnate alla macchina virtuale Vagrant sono quelle riportate nella Tabella 4.4.

	Configurazione macchina Vagrant
S.O	Ubuntu 14.04.5 LTS 64-bit
CPU	2
Memoria RAM	6 GB
Memoria RAM libera	4,6 GB
Disco	40 GB

Tabella 4.4: Configurazioni della macchina virtuale Vagrant nella seconda soluzione

In questo caso la saturazione delle risorse avverrà sulla macchina virtuale con l'andamento riportato nella Tabella 4.5.

N Repliche attive	1	3	5	10
Percentuale utilizzo CPU	2%	2%	2.5%	3%
Memoria RAM utilizzata	1,7 GB	1,84 GB	1,90 GB	1,99 GB
Tempo di avvio Repliche	3sec	5 sec	12 sec	25 sec

Tabella 4.5: Risorse utilizzate al variare del numero di repliche attive

Come si può notare il numero di repliche attive potenziali risulta di molto superiore in questa soluzione. A parità di numero di repliche attive le risorse utilizzate in questo caso sono decisamente minori rispetto alla soluzione precedente. Anche per quanto riguarda i tempi di avvio dell'applicazione si ha un deciso miglioramento con lo swarm di Docker risultando molto più rapidi rispetto alla prima soluzione. In questo caso, inoltre, si deve attendere l'avvio di una sola macchina Vagrant e l'avvio dello swarm all'interno di questa è pressoché immediato. Inoltre non è necessario attendere che tutte le copie siano attive per poter iniziare ad utilizzare l'applicazione, mentre nel primo caso si deve attendere il completamento dell'operazione di avvio di tutte le macchine virtuali.

Nel grafico in Figura 4.3 è riportato un confronto tra le due soluzioni, in termini di risorse utilizzate al crescere del numero di repliche. La curva in blu rappresenta l'andamento del consumo di memoria RAM al crescere del numero di repliche attive nella prima soluzione. La curva rossa invece identifica il consumo della memoria RAM nella seconda soluzione. Come si può notare nella prima soluzione si ha una crescita lineare nel consumo delle risorse. Nella seconda soluzione invece abbiamo un andamento di tipo logaritmico. I dati ottenuti ci confermano quanto ipotizzato: la soluzione che utilizza Docker permette di avere in esecuzione in contemporanea più copie dell'applicazione Cyber-gym rispetto alla soluzione con macchi-

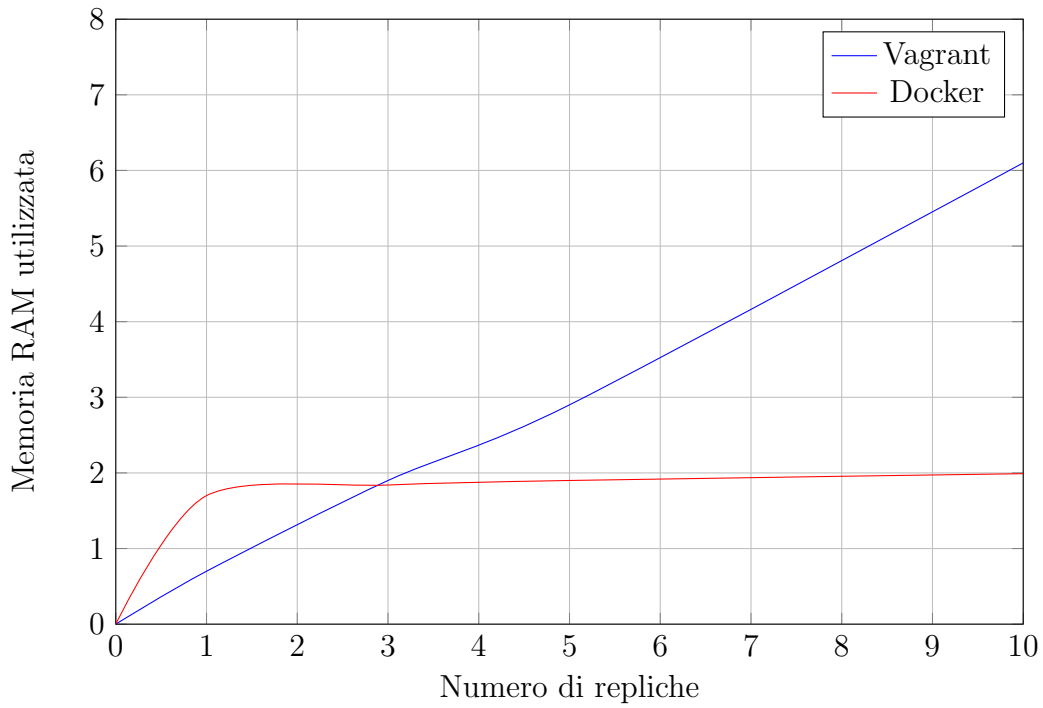

Figura 4.3: Confronto tra i risultati ottenuti

ne Vagrant. In modo duale, questi risultati ci dicono che tramite Docker Swarm possiamo ottenere lo stesso numero di repliche utilizzando molte meno risorse rispetto a Vagrant.

Ovviamente i numero massimo di repliche ottenibili e le prestazioni stesse dei sistemi descritti possono subire delle variazioni in base alla macchina fisica che viene utilizzata e ai processi in esecuzione su di essa, tuttavia la differenza tra le due soluzioni rimane inalterata.

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi si è voluto proporre e realizzare un sistema di sandboxing per servizi di web security training. In particolare sono state realizzate due versioni di questo sistema, utilizzando due metodi di virtualizzazione differenti, Vagrant e Docker. Entrambi i metodi sono stati analizzati approfonditamente per poterne sfruttare al meglio le caratteristiche principali. Su tutte e due le versioni del sistema di sandboxing sono stati effettuati dei test per valutarne le prestazioni e il consumo di risorse sulla macchina fisica. I risultati di questi test hanno evidenziato che la soluzione realizzata sfruttando le caratteristiche del servizio Docker risulta più efficiente dal punto di vista delle prestazioni e presenta una maggiore scalabilità rispetto alla soluzione basata su Vagrant. Entrambe le versioni del sistema sono state realizzate a partire dall'applicazione di web security training chiamata Cyber-gym.

Le tecnologie di virtualizzazione utilizzate sono in continua evoluzione, pertanto è possibile pensare di ottimizzare ulteriormente le performace del sistema proposto anche testandolo su hardware più performante e integrandolo con nuove tecnologie. Un ulteriore sviluppo futuro prevede di

applicare, con le opportune modifiche, tale sistema ad altre applicazioni di web security training, anche con funzionalità aggiuntive rispetto a quella utilizzata come caso d'uso per questa tesi.

Appendice A

Script proxy server

A.1 Prima soluzione

Script del proxy server per la prima soluzione

```
1  var args = process.argv.splice(2);
2  var vagrant = require('vagrant');
3  var httpProxy = require('http-proxy');
4  var http = require('http');
5  var express = require('express');
6  var app = express();
7  var jsonObj = require("./cyber-gym/Machines.json");
8  var iterator = 1;
9
10
11
12 vagrant.up(function(code) {
13     console.log('Cyber-gym is ready');
14 });
```

```
15
16 app.get('/', function (req, res) {
17 var target_address = getTargetAddress();
18   proxy.web(req, res, { target: target_address });
19
20 });
21
22 app.get('/get_status', function(req, res){
23   console.log('get status');//callback
24   vagrant.status(function(code){
25    cosole.log('status');
26   });
27   res.end();
28 });
29
30 app.get('/restart_machine',function(req,res){
31   var oldID = iterator;
32   var oldAddress = jsonObj['Address'+iterator];
33   var oldName = jsonObj['Machine'+iterator];
34
35   delete jsonObj['Address'+iterator];
36   delete jsonObj['Machine'+iterator];
37   if (iterator == 4 ) {
38     iterator = 1;
39   }
40   else {
41     iterator++;
```

```
42    }
43  res.end();
44  restartmachine(oldID, oldAddress, oldName);
45  });
46
47
48  function restartMachine(id, address, name){
49  vagrant.destroy(name, function(code){
50    console.log('Machine destroyed');
51  });
52  vagrant.up(name,function(code){
53    addMachineToList(id, address, name);
54    console.log('Machine ready');
55  });
56
57  }
58  function addMachineToList(id, address, name){
59    var addresskey = 'Address'+id;
60    var namekey = 'Machine'+id;
61    jsonObj[namekey]=name;
62    jsonObj[addresskey]=address;
63    return;
64
65  }
66
67  function getTargetAddress(jsonObj, iterator){
68
```

```
69    return jsonObj['Address'+iterator];
70
71  }
72
73
74  var proxy = httpProxy.createProxyServer({});
75  var server = http.createServer(app, function(req, res
        ) {});
76  server.listen(8080);
```

A.2 Seconda soluzione

Script del proxy server per la seconda soluzione

```
1  var args = process.argv.splice(2);
2  var vagrant = require('vagrant');
3  var httpProxy = require('http-proxy');
4  var http = require('http');
5  var express = require('express');
6  var app = express();
7  var jsonObj = require("./cyber-gym/Services.json");
8  var iterator = 1;
9
10
11
12 vagrant.up(function(code) {
13     console.log('Cyber-gym is ready');
14 });
15
16 app.get('/', function (req, res) {
17 var target_address = getTargetAddress();
18   proxy.web(req, res, { target: target_address });
19
20 });
21
22 app.get('/restart_service',function(req,res){
23   var oldID = iterator;
24   var oldPort = jsonObj['Port'+iterator];
```

```
25    var oldName = jsonObj['Service'+iterator];
26
27    delete jsonObj['Port'+iterator];
28    delete jsonObj['Service'+iterator];
29    if (iterator == 6 ) {
30      iterator = 1;
31    }
32    else {
33      iterator++;
34    }
35  res.end();
36  restartService(oldID, oldPort, oldName);
37  });
38
39
40  function restartService(id, port, name){
41  vagrant.ssh('virtual -c sudo docker remove' + name,
        function(code){});
42
43  vagrant.ssh('virtual -c sudo docker-compose.yml' +
        name, function(code){});
44  }
45
46  }
47  function addServiceToList(id, port, name){
48    var portkey = 'Address'+id;
49    var namekey = 'Service'+id;
```

```
50    jsonObj[namekey]=name;
51    jsonObj[Portkey]=port;
52    return;
53
54 }
55
56 function getTargetAddress(jsonObj, iterator){
57
58    var Port = jsonObj['Port'+iterator];
59
60 }
61
62
63 var proxy = httpProxy.createProxyServer({});
64 var server = http.createServer(app, function(req, res
       ) {
65 });
66 server.listen(8080);
```

Bibliografia

[1] Fielding R. T. Architectural styles and the design of network-based software architectures. 2000, Universty Of California.

[2] Owasp Top Ten, https://www.owasp.org/index.php.

[3] LeBlanc D. Howard M. *Writing secure code*. 2002.

[4] Clarke J. *SQL Injections Attacks and Defense*. 2012.

[5] Vagrant, https://www.vagrantup.com.

[6] Hashimoto M. *Vagrant up and running*. 2013.

[7] Scotchbox.io, https://box.scotch.io/.

[8] Docker, https://www.docker.com.

[9] Ferreira A. Rajamony R. Rubio J. Felter, W. An updated performance comparison of virtual machine and linux containers. (ISPASS), 2015 IEEE International Symposium On (2015).

[10] Alessandro Orso William G.J. Halfond, Jeremy Viegas. A classification of sql injection attacks and countermeasures.