UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

CORSO DI APPLICAZIONI TELEMATICHE

CORSO DI NETWORK SECURITY

DOCKER HACKING TOOLS

DOCUMENTAZIONE ESTERNA

Studente:
Pierpaolo PICCIRILLO

Professore:
Simon Pietro ROMANO

ANNO ACCADEMICO 2018-2019

Indice

1

Introduzione

Docker Hacking Tools è una web application che consente di lanciare e controllare molteplici container docker in esecuzione su un server. Nello specifico, tutti i container sono lanciati con l'opzione *–rm*, il che consente di eseguire un qualsiasi tool (per il quale esista una docker image associata) alla stregua di un comando eseguito in un terminale.

1.1 Prerequisiti

Gli unici prerequisiti richiesti per il funzionamento dell'applicazione sono:

- il **docker engine**: necessario lato server per l'avvio, l'esecuzione e la gestione dei vari container; disponibile gratuitamente sul sito ufficiale[1].
- un **browser web** qualsiasi: necessario lato client per l'interfacciamento con le funzionalità offerte.

1.2 Getting started

Spostarsi nella cartella *server*, installare le dipendenze del back-end ed avviarlo:

```
sudo npm install && sudo node server.js
```

Quindi spostarsi nella cartella *client*, installare le dipendenze del front-end ed avviarlo:

```
sudo npm install && ng serve
```

L'interfaccia è raggiungibile all'indirizzo

```
localhost:4200
```

[1] https://hub.docker.com/search/?type=edition&offering=community

2

Modello architetturale

Docker Hacking Tools è una web application sviluppata conformemente allo stile architetturale Representational State Transfer (REST), molto comune nell'ambito dei sistemi distribuiti.

Nello specifico, sono stati sviluppati due sotto-sistemi che utilizzano un sistema di comunicazione completamente basato su HTTP e per il quale non sono stati previsti meccanismi di autenticazione e di protezione del canale.

Questi due sotto-sistemi consistono in:

- un **back-end** sviluppato tramite il noto framework web *Express* basato sul runtime di JavaScript *Node.js*. Esso risiede sulla stessa macchina su cui è presente il docker engine, e si interfaccia con esso tramite la libreria *docker-js*[2].
- un **front-end** sviluppato tramite la piattaforma *Angular* che gestisce l'interazione con l'utente e le funzionalità implementate dal back-end.

Nel seguito si riporta una descrizione dettagliata dei due sotto-sistemi e dell'API REST offerta dal back-end, nonchè la documentazione relativa al file di configurazione *myconfig.json* (che offre la possibilità di semplificare il setup di comandi prefissati da eseguire).

Nota sulla documentazione Il presente documento rappresenta la documentazione esterna del software. Gli aspetti puramente implementativi non sono stati qui riportati perchè analizzati nella documentazione interna (commenti nel codice).

[2] https://github.com/giper45/docker-js

2.1 Back-end in Node.js/Express

Il back-end dell'applicazione, come detto, è stato realizzato su uno stack costituito da *Node.js* ed *Express*. Il primo offre prestazioni elevate e scalabili grazie al modello architetturale basato sul concetto di event-loop, mentre il secondo consente di ridurre drasticamente la complessità di sviluppo di un web server.

Da un punto di vista puramente implementativo, il back-end consiste in un unico file server.js, che si occupa di:

- dichiarare le **dipendenze** e le librerie utilizzate:

```
1 const express = require('express');
2 const cors = require('cors');
3 const bodyParser = require('body-parser');
4 const fse = require('fs-extra');
5 const _ = require('underscore');
6 const multipart = require('connect-multiparty');
7 const myDockerLib = require('./lib/myDockerLib');
```

  Di particolare interesse risultano essere la *multipart*, che agisce da middleware tra le richieste HTTP POST e l'implementazione del servizio associato occupandosi del download di eventuali file inviati dal front-end; e la ***myDockerLib***, che implementa:

  - *myRun()*: una funzione di *run* dei container potenziata rispetto a quella presente in *docker-js*, in quanto gestisce anche i parametri relativi all'opzione –rm e quella relativa al mount dei volumi (fino a 2);
  - *myGetContainerInfo(), myStopContainer()*: funzioni di supporto all'esecuzione delle funzionalità mappate sulle richieste HTTP di tipo GET, PUT e DELETE;

- **avviare** il web server (sulla porta 3000);
- mappare le **rotte HTTP** sulle relative funzionalità (dettagliate nel paragrafo relativo alle API REST).

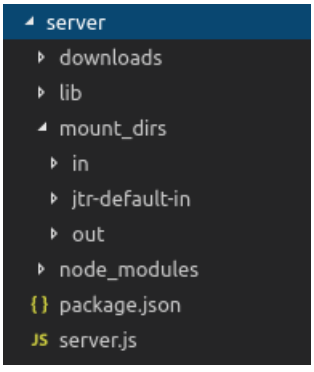
Inoltre, il server mantiene internamente una lista di container (*containerList* nel codice) in esecuzione (*running*) o interrotti (*exited*): si noti che nel secondo caso, l'informazione è necessaria per la gestione dei volumi di mount ancora presenti su disco (infatti, data l'esecuzione della *docker run* in modalità *–rm*, si ha che il relativo container è già stato rimosso in precedenza dal docker engine). Si noti anche che almeno un volume di mount è sempre presente, anche

se non esplicitamente richiesto dall'utente all'atto di sottomissione nel front-end: esso è necessario per il salvataggio dell'output della console del container avviato, e che potrà essere restituito al front-end fino ad una sua esplicita richiesta di cancellazione.

Al momento dell'avvio del back-end e più volte durante l'esecuzione, viene consultato il file di configurazione *myconfig.json* (trattato nel dettaglio più avanti), in modo tale da precaricare configurazioni di default e di preset per docker image specifiche. Una volta avviato il back-end, esso mostrerà a schermo le configurazioni di default (di seguito riportate), nonchè il timestamp e il tipo di richiesta all'atto di ricezione di una richiesta HTTP.

```
pic@PP-PC ~/Scrivania/progetto/server sudo node server.js
Server configurato con:
  server_mount_dirs_in: /home/pic/Scrivania/progetto/server/mount_dirs/in
  server_mount_dirs_out: /home/pic/Scrivania/progetto/server/mount_dirs/out
  downloads_dir: /home/pic/Scrivania/progetto/server/downloads
  out_dir_container: /home
  default_append_cmd: /home/out_console.txt
Server avviato sulla porta 3000...
    [1559380818024] POST /api/test1
    [1559380818836] container test1 avviato con successo
    [1559380819972] GET /api/
    [1559380832486] POST /api/test2
    [1559380833280] container test2 avviato con successo
    [1559380834424] GET /api/
    [1559380836438] GET /api/test1
    [1559380836444] GET /api/test2
```

Infine, si riporta la struttura gerarchica del back-end:

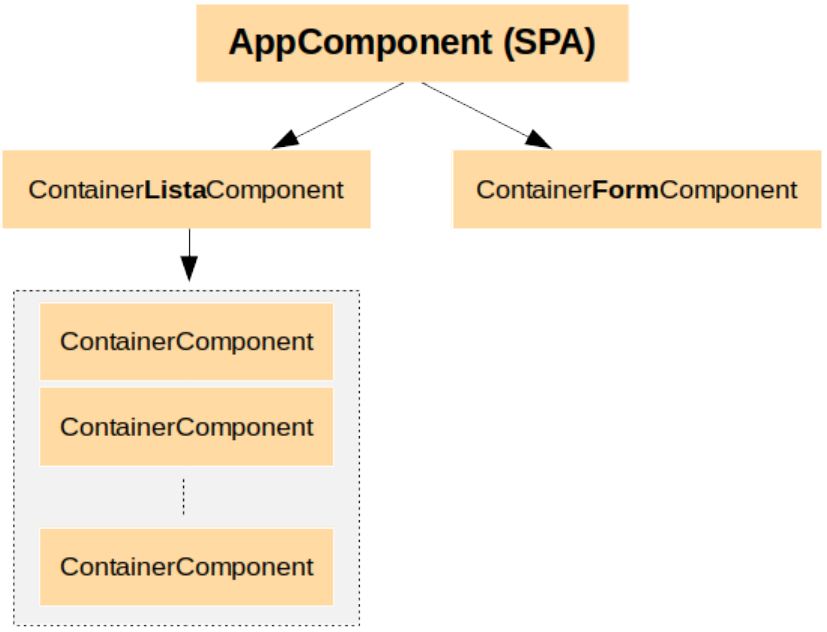


Sono di interesse le directory *downloads*, nella quale viene effettuato dal middleware *multipart* il download dei file inviati dal front-end, e *mount-dirs*: in quest'ultima sono presenti eventuali cartelle associate a dei preset contenuti nel *myconfig.json* (è questo il caso della *jtr-default-in*), nonchè le cartelle *in* e *out* nelle quali vengono creati i volumi di mount dei vari container (quello di *in* viene inizializzato con i file scaricati in *downloads* e con quelli presenti nelle directory di preset).

2.2 Front-end in Angular

Il front-end dell'applicazione è stato realizzato tramite la piattaforma *Angular*. In tal modo si è massimizzata la modularità del codice realizzando una moderna web app in stile Single-Page Application (SPA), in cui logiche di presentazione e di business sono fortemente disaccoppiate (conformemente al pattern architetturale *Model-View-Whatever* o *MV**, evoluzione del classico *Model-View-Controller* o *MVC*).

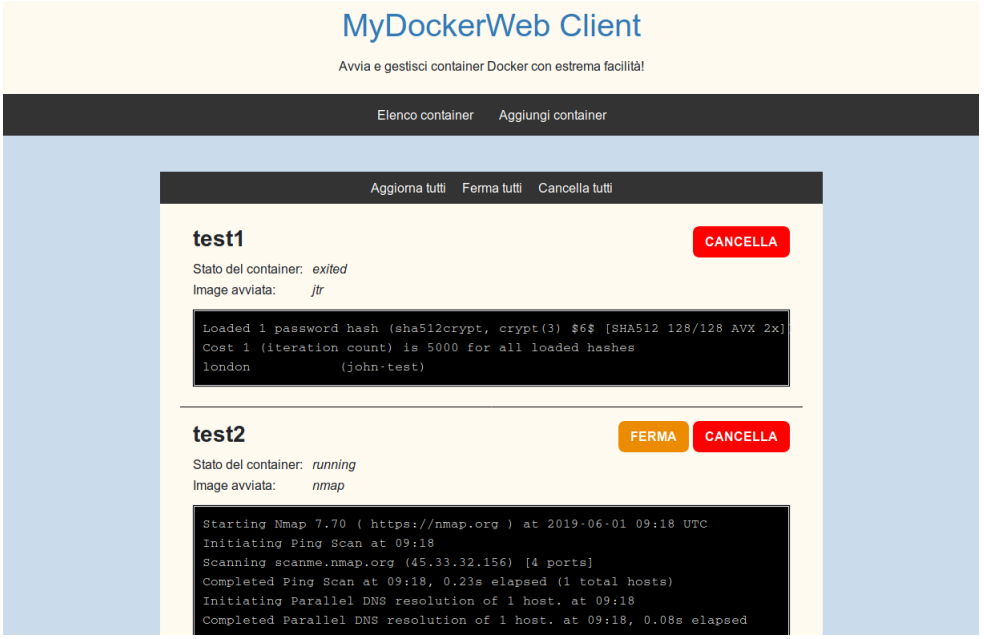
Da un punto di vista architetturale, è presente l'*app.module* tipico di qualsiasi applicazione *Angular*: esso effettua il bootstrap con l'*app.component* che funge da "guscio" (la SPA a cui si è accennato precedentemente) per l'esecuzione di tutti gli altri componenti applicativi. Si riporta di seguito uno schema gerarchico dei vari componenti:



Tramite il *RouterModule* di *Angular*, in base al path del browser è possibile cambiare dinamicamente il contenuto dell'*app.component*, che può renderizzare in mutua esclusione:

- il *container-lista.component*: è a sua volta composto da zero o più *container.component*. Esso, tramite richiesta HTTP GET a livello di collezione, ottiene la lista e i dati associati a tutti i container *running* o *exited* presenti sul back-end: in questo modo è in grado di costruire oggetti *Container* e di iniettarli nei singoli *container.component*. Ciascun *container.component* si occupa di renderizzare a video l'output della console e di aggiornarlo tramite un meccanismo di polling (con intervallo di 2 secondi tra le richieste) fin-

chè il container associato non termina (ossia risulta che la proprietà *running* associata all'oggetto diventa *false*);



- il *container-form.component*: consente di sottomettere una nuova istanza di container al back-end. La form è stata sviluppata secondo l'approccio Reactive Form di *Angular*, ossia viene costruita e validata tramite codice TypeScript (piuttosto che direttamente nel template HTML). Inoltre, il componente si serve del file di configurazione *myconfig.json* per il preset automatico di alcuni campi della form.

Infine, è stato implementato anche un *container-rest.service*, un servizio che consente l'interfacciamento tramite API REST con il back-end: il *container-lista.component* si serve delle primitive a livello di collezione, il *container.component* di quelle a livello di elemento.

2.3 API REST

La comunicazione tra back-end e front-end avviene tramite HTTP mediante l'implementazione di API REST. L'endpoint su cui esse sono disponibili è *http://<back-end-ip>:3000/api*, ed è possibile usufruire di funzionalità a livello di collezione e a livello di singolo elemento.

2.3.1 Accesso a singolo elemento

Per accedere ad un singolo elemento, l'endpoint generale precedentemente riportato viene esteso con il nome associato al container di interesse: se ad esempio si vogliono ottenere le informazioni aggiornate relative al container con nome *test1*, allora verrà effettuata una *GET http://<back-end-ip>:3000/api/test1*.
I metodi HTTP mappati sono:

- **GET**: restituisce un JSON (mappabile sull'oggetto Container da parte del front-end), oppure un codice di errore 404 con relativo messaggio se il container non è stato trovato. Un esempio di oggetto restituito:

```
1 {
2     "name": "test1",
3     "running": false,
4     "image": "jtr",
5     "outConsole": "Loaded 1 password hash (sha512crypt, crypt
          (3) $6$ [SHA512 128/128 AVX 2x])\nCost 1 (iteration
          count) is 5000 for all loaded hashes\nlondon          (
          john-test)\n"
6 }
```

- **POST**: consente di avviare un nuovo container sul back-end, specificando se l'image da avviare fa parte di un preset contenuto nel *myconfig.json* oppure se si tratta di un'image custom specificata dall'utente (nel caso in cui essa non fosse presente nel repository del docker engine, viene scaricata automaticamente). Essendo la POST di tipo multi-part, è possibile anche effettuare l'upload di file dal front-end al back-end. La risposta può essere di 3 tipi:

- codice di successo 202 con relativo messaggio se il container viene avviato correttamente;
- codice di errore 409 con relativo messaggio se il container non può essere avviato a causa di un container già presente con lo stesso nome richiesto;
- codice di errore 503 con relativo messaggio se il back-end non può soddisfare la richiesta (ad esempio, perchè il docker engine non risponde).

- **PUT**: consente di fermare un container attualmente in esecuzione sul back-end, senza rimuovere dal filesystem i mount associati (quindi ad esempio sarà ancora possibile visualizzare l'output generato dal container fino a quel momento). Si noti che l'istanza della image in docker viene rimossa automaticamente in quanto all'avvio viene specificata l'opzione *–rm*. Restituisce un codice di successo 202 con relativo messaggio in caso di esito positivo, o un codice di errore 404 con relativo messaggio se il container con il nome specificato non viene trovato.

- **DELETE**: consente di fermare un container (se attualmente in esecuzione sul back-end) e di rimuovere dal filesystem i mount associati (a differenza della PUT). Restituisce un codice di successo 202 con relativo messaggio in caso di esito positivo, o un codice di errore 404 con relativo messaggio se il container con il nome specificato non viene trovato.

2.3.2 Accesso alla collezione di elementi

Utilizzando l'endpoint generale precedentemente riportato senza specificare alcun nome, è possibile applicare i metodi HTTP documentati nel paragrafo precedente a tutti i container gestiti dal back-end. Ad esempio, se si vogliono ottenere le informazioni aggiornate relative a tutti container, allora verrà effettuata una *GET http://<back-end-ip>:3000/api/*. Un possibile risultato è di seguito riportato (JSON array):

```
1 [
2     {
3         "name": "test1",
4         "running": true,
5         "image": "jtr",
6         "outConsole": "Loaded 1 password hash (sha512crypt, crypt(3)
              $6$ [SHA512 128/128 AVX 2x])\nCost 1 (iteration count)
             is 5000 for all loaded hashes\n"
7     },
8     {
9         "name": "test2",
```

```
        "running": false,
        "image": "custom",
        "outConsole": "helloworld\n"
    }
]
```

L'unico metodo non implementato a livello di collezione è POST in quanto non avrebbe senso se applicato a più container contemporaneamente.

2.4 File di configurazione

Il file *myconfig.json* conferisce all'applicazione estrema configurabilità e facilità d'uso, in quanto consente di creare una serie di preset associati a specifiche docker image al fine di ridurre al minimo i parametri da inserire/modificare all'atto di sottomissione di un nuovo container. Di seguito si riporta il contenuto attuale del file al fine di rendere più agevole la comprensione della sua struttura:

Listing 1 File di configurazione *myconfig.json*

```
{
    "serverGlobalConfig": {
        "in-dir": "/mount_dirs/in",
        "out-dir": "/mount_dirs/out",
        "downloads-dir": "/downloads",
        "out-dir-container": "/home",
        "to-append-cmd": "/home/out_console.txt"
    },

    "images": [
        {
            "name":"jtr",
            "image":"knsit/johntheripper",

            "serverConfig": {
                "runParams": {
                    "detached": true,
                    "rm": true,
                    "in-dir-container": "/in",
                    "out-dir-container": "/home/john"
                },
                "default-dir-to-copy-in-mount-in": "/mount_dirs/jtr-
                    default-in",
                "default-script-in": "init.sh",
                "to-append-cmd": "/home/john/out_console.txt"
            },
```

```
27              "client-preset": {
28                  "cmd-presets": [
29                      {"name":"wordlist", "cmd": "john --format=
                            sha512crypt --wordlist=/in/password.lst /in/
                            unshadowed"},
30                      {"name":"single crack", "cmd": "john --format=
                            sha512crypt --single /in/unshadowed"},
31                      {"name":"incremental", "cmd": "john --format=
                            sha512crypt --incremental /in/unshadowed"},
32                      {"name":"custom", "cmd": "john "}
33                  ],
34                  "mount-in-container": "/in"
35              }
36          },

38          {
39              "name":"nmap",
40              "image":"instrumentisto/nmap",

42              "serverConfig": {
43                  "runParams": {
44                      "detached": true,
45                      "rm": true,
46                      "in-dir-container": "/in"
47                  }
48              },

50              "client-preset": {
51                  "cmd-presets": [
52                      {"name":"scan TCP", "cmd": "nmap -v
                            scanme.nmap.org"},
53                      {"name":"custom", "cmd": "nmap "}
54                  ]
55              }
56          },

58          {
59              "name":"custom"
60          }
61      ]
62  }
```

Il primo oggetto presente è *serverGlobalConfig*: esso specifica alcuni settaggi di default per il back-end (che vengono stampati a video nel momento in cui il server viene avviato).

Il secondo oggetto presente è *images*: oltre alla configurazione di default con nome **custom** che consente all'utente di specificare nella form in *Angular* il nome della docker image da avviare, sono presenti due image di preset (a cui se ne possono aggiungere altre): *jtr* e *nmap*. Come è facilmente intuibile, la prima avvia una image con il tool di cracking delle password *John The Ripper*, mentre la seconda avvia una image con il tool di scanning di rete *nmap*. Per ciascuna di esse sono presenti due ulteriori sezioni:

- *serverConfig*: contiene dei settaggi per il back-end specifici dell'image. Ad esempio, *runParams* specifica i flag di default da passare a docker nel momento in cui si richiede la run del container; *to-append-cmd* sovrascrive il path di default (contenuto in *serverGlobalConfig*) per la localizzazione del file con l'output del container.
- *client-preset*: contiene i preset utilizzati dal front-end per pre-compilare i campi della form all'atto di sottomissione di un nuovo container. Ciò consente di risparmiare tempo e semplificare l'avvio di comandi uguali o simili.

Si noti che al momento il file *myconfig.json* è disponibili a entrambi i sotto-sistemi, in una versione futura dell'applicazione sarebbe opportuno che il front-end scaricasse dinamicamente il file dal back-end.

3

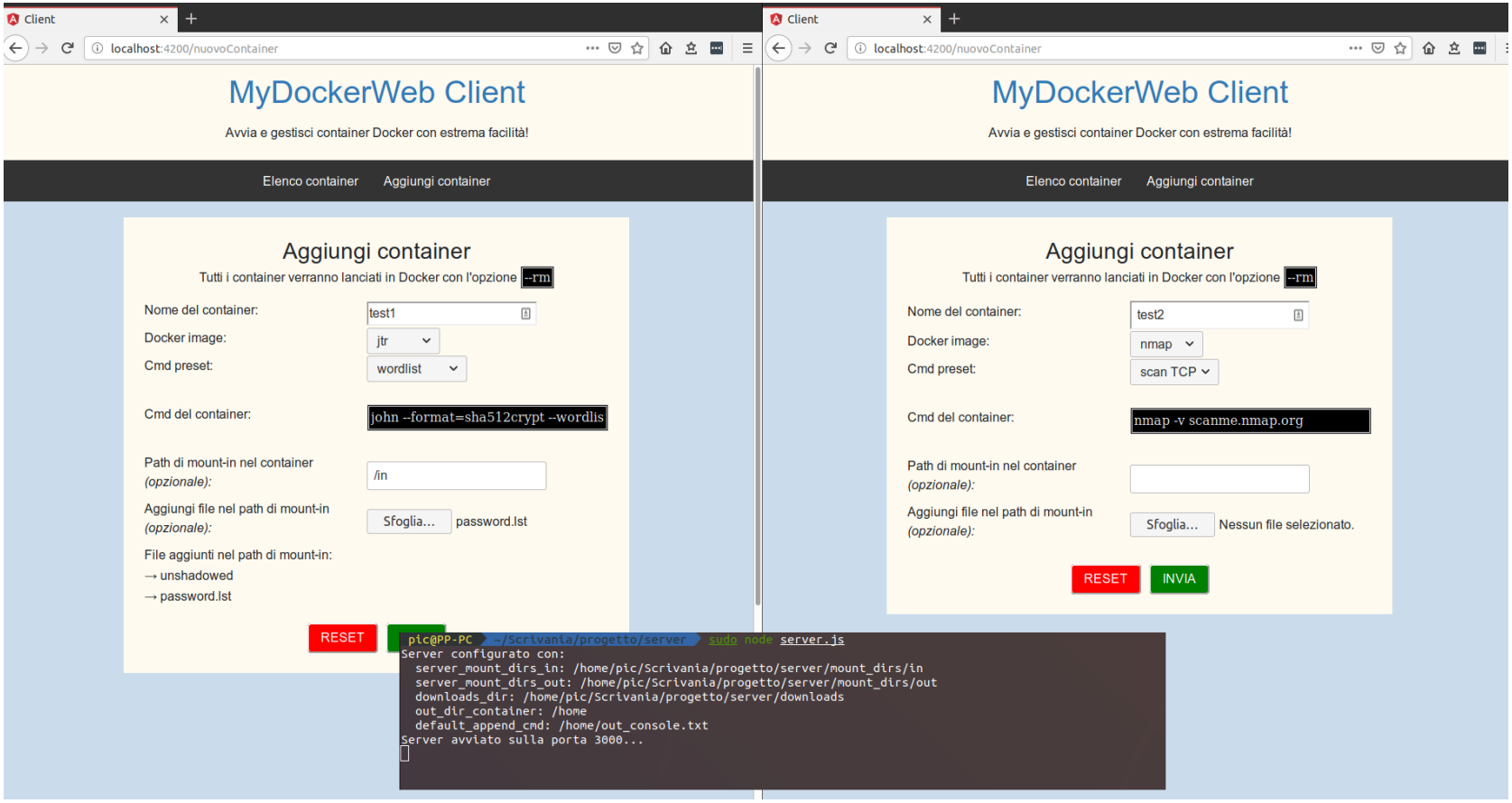
Esempio d'uso

Si riporta un esempio d'uso in cui si avviano due container con le seguenti image:

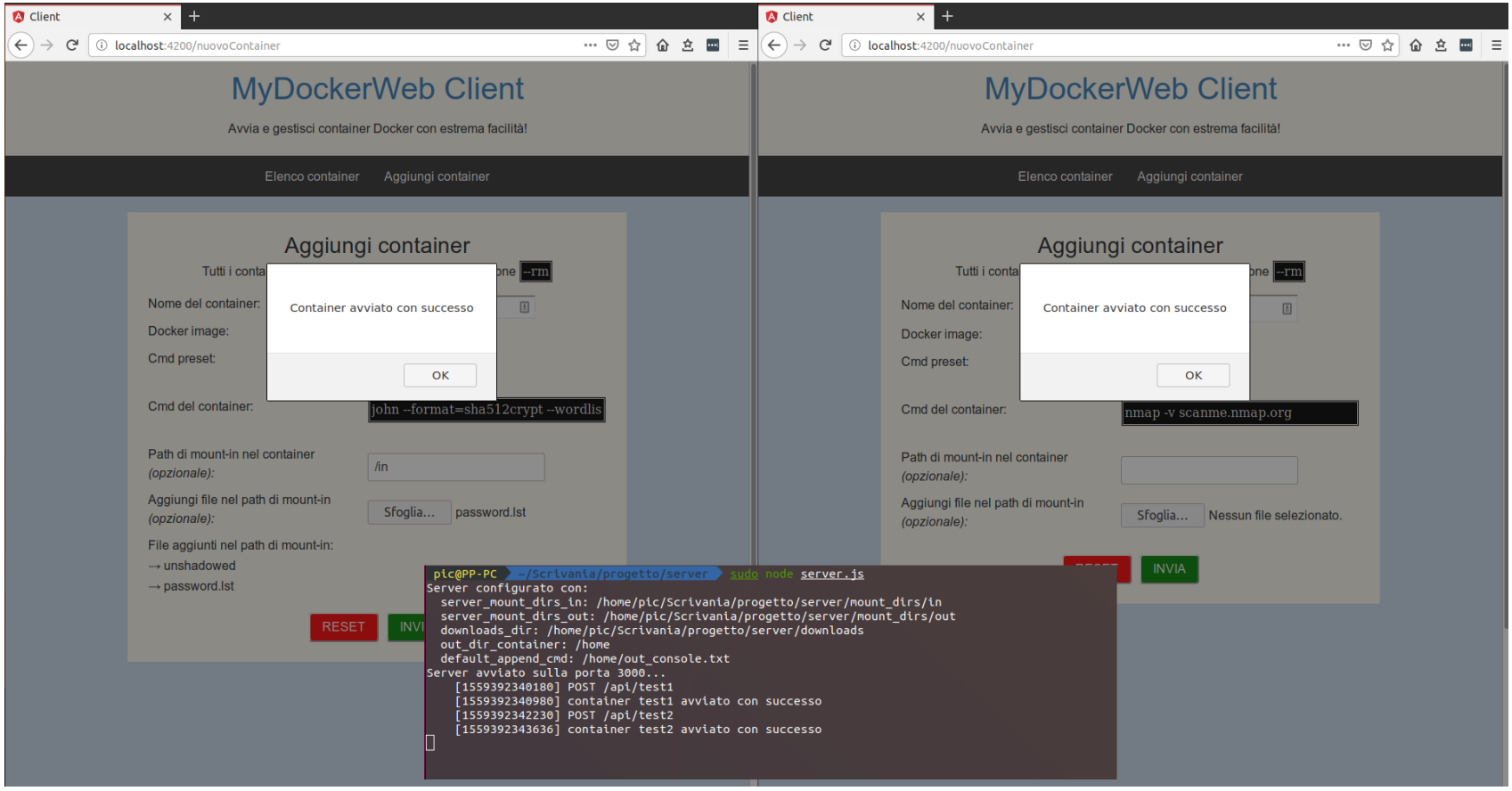
- *jtr*: preset che effettua un attacco a dizionario di una password (inviata in upload);
- *nmap*: preset che effettua uno scan di un sito di test.

Si riporteranno una serie di screenshot in cui è visibile l'interfaccia utente (front-end) e il terminale nel quale è in esecuzione il back-end.

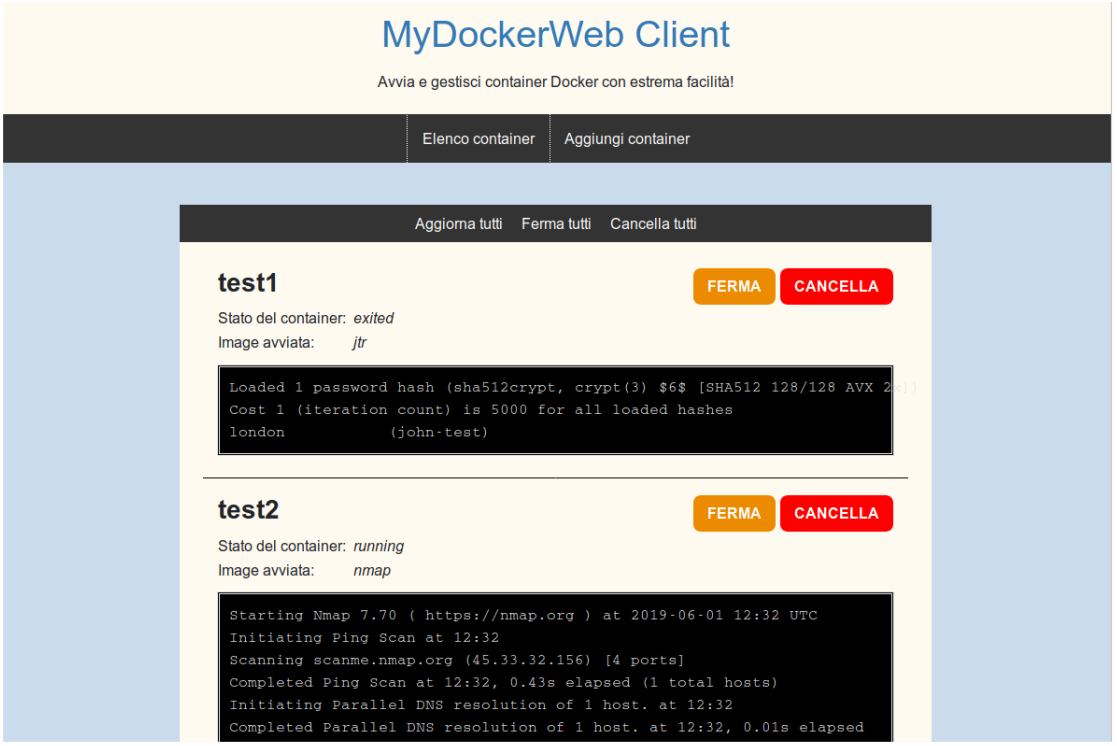
1. Compilazione delle form di sottomissione Tramite la form del front-end si selezionano di due preset desiderati (si noti anche come per jtr siano stati selezionati due file da caricare):



2. Avvio dei container Si inviano le form al back-end che mostra a video un log relativo alla ricezione della richiesta, invece il front-end notifica l'esito della sottomissione tramite un alert:



3. Visualizzazione dei container Il front-end effettua una prima richiesta GET all'inizio per ottenere la lista dei container presenti sul back-end, dopodichè effettua richieste GET ogni 2 secondi per ciascun container ancora in esecuzione (*running*). Sul back-end vengono loggate a video tutte le richieste:




```
pic@PP-PC ~/Scrivania/progetto/server sudo node server.js
Server configurato con:
  server_mount_dirs_in: /home/pic/Scrivania/progetto/server/mount_dirs/in
  server_mount_dirs_out: /home/pic/Scrivania/progetto/server/mount_dirs/out
  downloads_dir: /home/pic/Scrivania/progetto/server/downloads
  out_dir_container: /home
  default_append_cmd: /home/out_console.txt
Server avviato sulla porta 3000...
    [1559392340180] POST /api/test1
    [1559392340980] container test1 avviato con successo
    [1559392342230] POST /api/test2
    [1559392343636] container test2 avviato con successo
    [1559392354804] GET /api/
    [1559392356827] GET /api/test1
    [1559392356831] GET /api/test2
```

4. Rimozione dei container Sul front-end si clicca su *Cancella tutti* per eliminare i due container (solo *test2* è ancora in esecuzione), quindi viene inviata una DELETE a livello di collezione. Lato back-end, vengono fermati i container ancora in esecuzione, e vengono rimossi i mount associati:


```
pic@PP-PC ~/Scrivania/progetto/server sudo node server.js
Server configurato con:
  server_mount_dirs_in: /home/pic/Scrivania/progetto/server/mount_dirs/in
  server_mount_dirs_out: /home/pic/Scrivania/progetto/server/mount_dirs/out
  downloads_dir: /home/pic/Scrivania/progetto/server/downloads
  out_dir_container: /home
  default_append_cmd: /home/out_console.txt
Server avviato sulla porta 3000...
    [1559392340180] POST /api/test1
    [1559392340980] container test1 avviato con successo
    [1559392342230] POST /api/test2
    [1559392343636] container test2 avviato con successo
    [1559392354804] GET /api/
    [1559392356827] GET /api/test1
    [1559392356831] GET /api/test2
    [1559392358827] GET /api/test1
    [1559392358833] GET /api/test2
    [1559392360837] GET /api/test2
    [1559392362837] GET /api/test2
    [1559392364835] GET /api/test2
    [1559392366837] GET /api/test2
    [1559392368836] GET /api/test2
    [1559392370842] GET /api/test2
    [1559392372838] GET /api/test2
    [1559392374837] GET /api/test2
    [1559392375582] DEL /api/
    [1559392375592] GET /api/
```