INDICE

1 SISTEMA DDOS CON AWS

L'idea alla base del progetto è la realizzazione di un'infrastruttura di macchine che possa performare un attacco **DDoS** (Distributed Denial of Service) a discapito di un web server generico, il tutto sfruttando i meccanismi del cloud computing ed in particolare i servizi offerti da **AWS** (Amazon Web Services). La realizzazione del progetto è stata divisa in diversi task da completare per avere un approccio più organizzato allo sviluppo. I task individuati, che verranno trattati più nello specifico nella relazione sono:

- Studio, utilizzo e configurazione delle istanze **EC2** fornite da AWS
- Implementazione del web server
- Implementazione del software di HTTP-flood
- Creazione e configurazione della rete privata
- Configurazione e creazione della comunicazione da client malevolo e le istanze EC2

L'obiettivo finale del progetto è stato quello di riuscire a rendere inutilizzabile il servizio che esporta il web server grazie alla potenza computazionale fornita in prestito da AWS. Di seguito è riportata l'architettura che è stata implementata per realizzare il tutto:

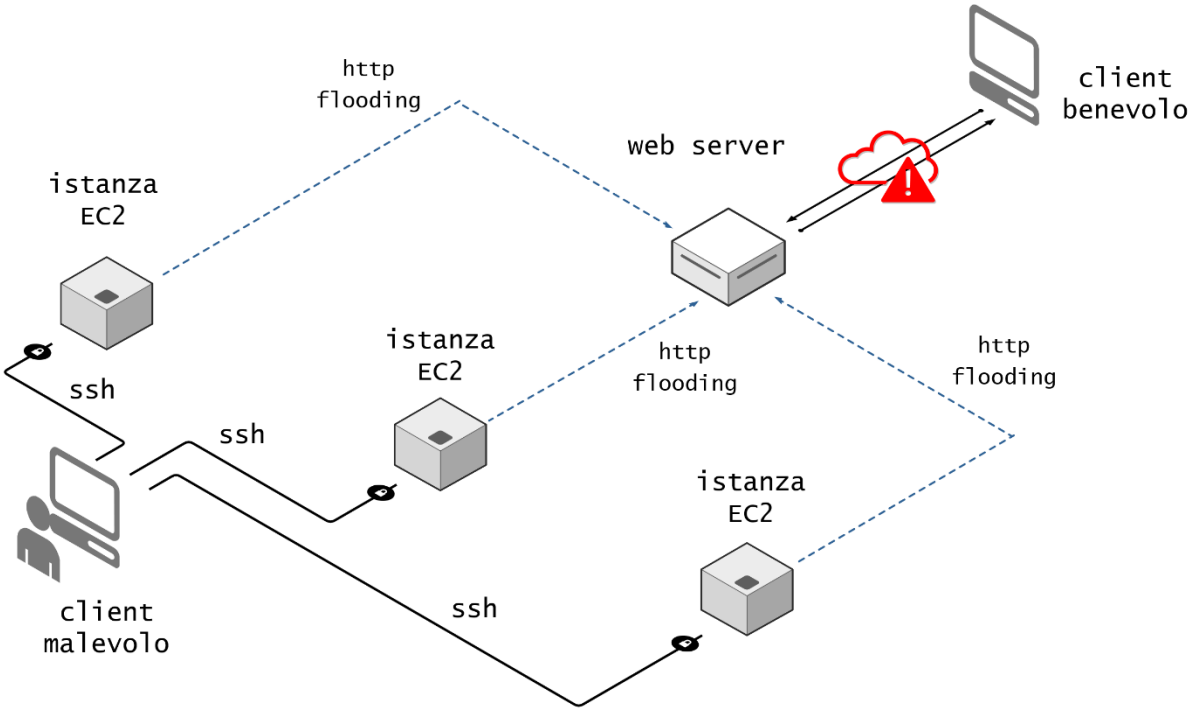


Figura 1: architettura del progetto

Nell'immagine vengono descritte le principali componenti del progetto. Abbiamo il **client malevolo** che tramite uno script (che sfrutta **ssh**) si collega alle istanze EC2 e fa partire l'attacco in parallelo su tutte le macchine. Le istanze prese in esempio sono 3 ma potenzialmente possono essere decine o centinaia di nodi, e tutti insieme inondano di

richieste HTTP un singolo web server obbligandolo a processare più richieste di quanto possa reggere. Il web server fornisce una web-app che viene simulata da un servizio molto semplice che ogni secondo effettua una richiesta HTTP al server per aggiornare un componente grafico innestato nella pagina sfruttando le chiamate asincrone in javascript. Facendo ciò viene simulato un qualsiasi servizio che necessita di iterazione con il server (fondamentale per accorgersi che qualcosa non va). Infine, abbiamo il client che simula un utilizzatore del servizio il quale, se l'attacco va a buon fine, non riesce più ad usufruire di quelle funzionalità ed avviene il crash del servizio con l'impossibilità di ricollegarsi alla pagina finché l'attacco non viene gestito o viene interrotto.

1.1 AWS E ISTANZE EC2

Nella prima fase del progetto ci si è incentrati sullo studio della piattaforma AWS e sull'utilizzo e la configurazione delle macchine EC2. In particolar modo è stato configurato l'ambiente delle istanze EC2 per rendere il tutto più agevole andando quindi a creare sostanzialmente 3 macchine identiche dal punto di vista del software.

//censored

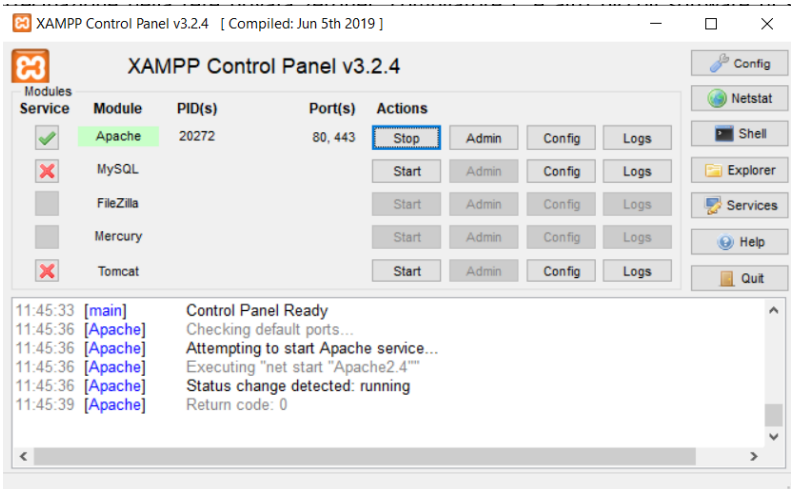
Sono state modificate le policy di sicurezza a tutte le macchine per permettere l'accesso remoto con GUI per sviluppare in maniera più agevole il software e per testare le comunicazioni con il web server sfruttando il protocollo RDP che viene utilizzato dal software **Remmina** proprio per permettere questa comunicazione.

//censored

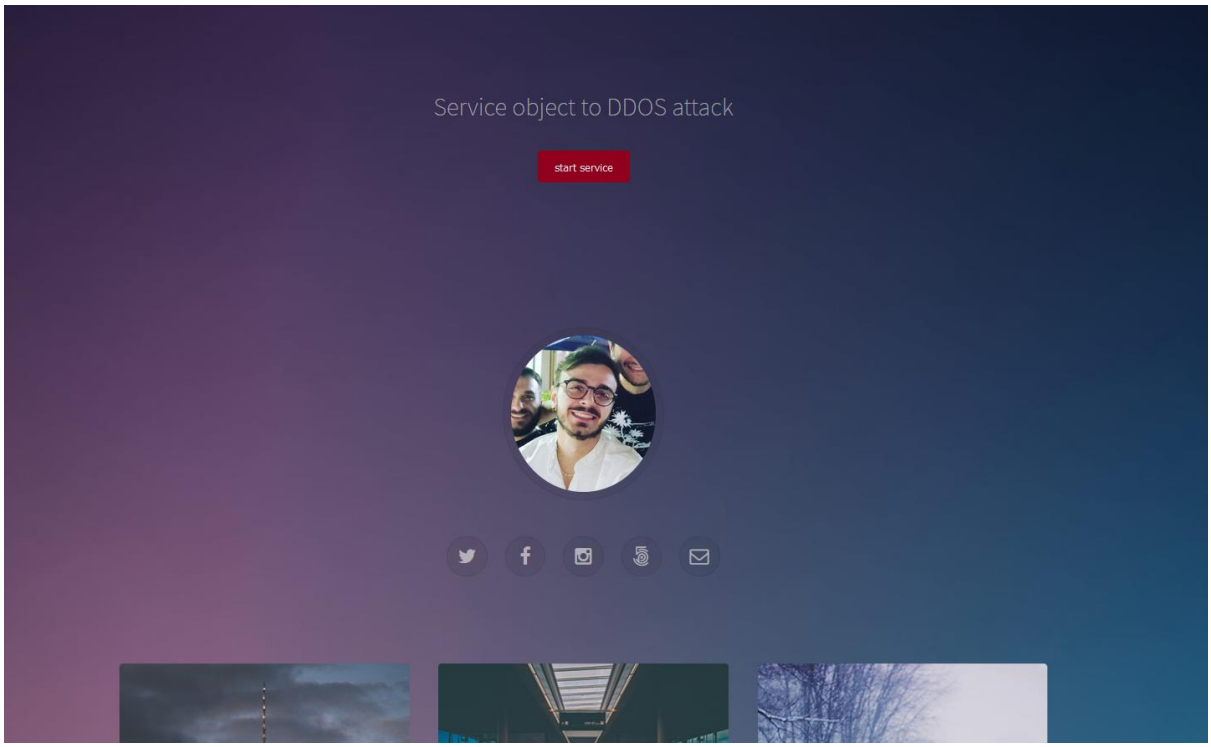
La scelta delle macchine è stata molto semplice in quanto, vincolati dai limiti dell'offerta free tier, l'unica scelta da prendere riguardava il sistema operativo e si è scelto di utilizzare istanze di Ubuntu 20.04 LTS in quando risultava essere compatibile con i software necessari per la comunicazione ed in generale per la sua compatibilità e la semplicità di utilizzo che contraddistingue Ubuntu da altre distribuzioni più complesse da utilizzare (per via del supporto al software di terze parti). La configurazione prevista per il funzionamento del tutto prevede ambienti di sviluppo come **Visual Studio**, software per il monitoraggio della rete, software per la creazione e partecipazione della rete privata **ZeroTierOne**, **compilatore gcc** e altri piccoli software di supporto come browser **Firefox** e **Remmina**.

1.2 WEB SERVER

Una volta configurate le macchine si è passati all'implementazione del web server. In questo caso il tutto è stato sviluppato con **XAMPP** (stack software di supporto allo sviluppo di applicazioni web locali) il quale permette di avere una collezione di strumenti quali **Apache**, **MariaDB**, **Php** e **Perl** che forniscono tutto il necessario per lo sviluppo web.



Dopo aver configurato il software XAMPP per permettere il collegamento tra le varie componenti si è passati allo sviluppo dell'applicazione web. In questo caso si è deciso di utilizzare un template come base per la pagina web modificando la parte javascript per implementare un semplice servizio che simula l'interazione con il web server. Tale servizio effettua delle richieste periodiche HTTP-GET al server obbligando l'utente ad instaurare una comunicazione per poter scaricare un'immagine che viene stampata nella pagina (come se fosse un caricamento). Esso simula inoltre lo scambio di informazioni che avviene nei tipici web server quando l'utente usufruisce di un servizio offerto dal provider.



Implementando il servizio in questo modo si è fatto in modo che l'attacco, qualora vada a buon fine, risulti essere ancora più "distruttivo" in quanto riesce a bloccare tale servizio oltre a rendere impossibile l'aggiornamento della pagina.

```
</head>
<body>
        <script src="assets/js/jquery.min.js"></script>
        <script src="assets/js/jquery.poptrox.min.js"></script>
        <script src="assets/js/skel.min.js"></script>
        <script src="assets/js/main.js"></script>
        <script type = "text/javascript" src="assets/js/myscript.js"></script>

        <div id="header">
        <div id ="demo">
        <h1> Service object to DDOS attack</h1>
        <button type="button" onclick="Tutor()"> start service </button>
        </div>
        </div>
```

Figura 2: estratto html della pagina web

```
<div id="demo">
<h1> the service is online as long as the wheel is turning </h1>
<ul class="icons">
<a>
<img src="assets/images/loading1.png" alt="" />
<br>
<button type="button" onclick="Stop()"> stop service </button>
</a>
</ul>
</div>
```

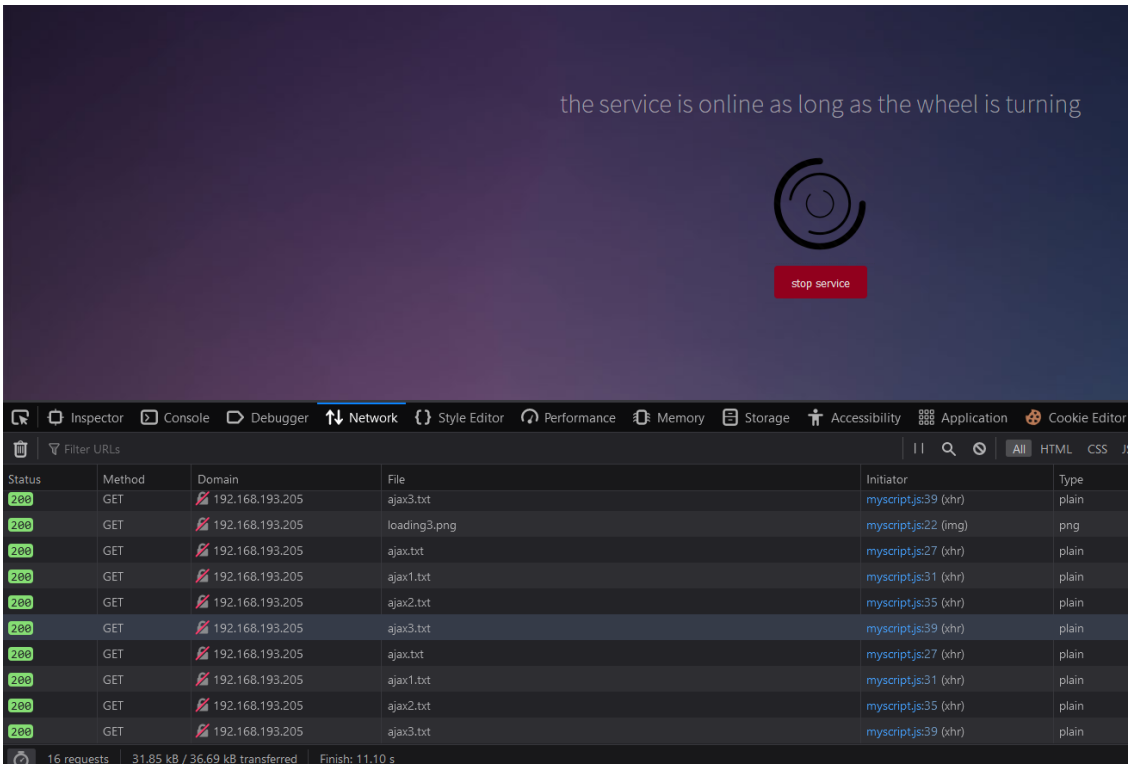
Figura 3: file ajax1.txt

```
function LoadDoc() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        document.getElementById("demo").innerHTML =
         this.responseText;}
    };
    if( cont === 0 ){
        xhttp.open("GET", "ajax.txt", true);
        cont = 1;
        xhttp.send();}
    else if (cont === 1){
        xhttp.open("GET", "ajax1.txt", true);
        cont = 2;
        xhttp.send();}
    else if (cont === 2){
        xhttp.open("GET", "ajax2.txt", true);
        cont = 3;
        xhttp.send();}
    else if (cont === 3){
        xhttp.open("GET", "ajax3.txt", true);
        cont = 0;
        xhttp.send();}
}

function Stop() {
    run = false;
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        document.getElementById("demo").innerHTML =
         this.responseText;}
    };
    xhttp.open("GET", "stop.txt", true);
    xhttp.send();}
```

Figura 4: estratto myscript.js

Da questa immagine si può vedere come il servizio generi le richieste HTTP che il server deve processare garantendo in questo modo un'interazione costante Client/Server.



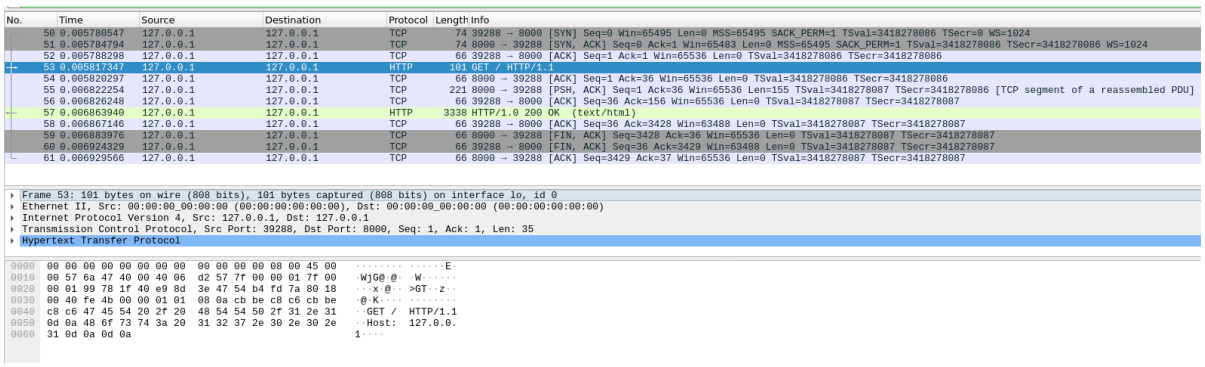
1.3 HTTP FLOODING

L'HTTP flooding attack è una tipologia di attacco DDoS (Distributed Denial of Service) dove un attaccante sfrutta delle richieste HTTP-GET/POST semi-legittime per inondare un web server o un'applicazione. Nel caso del progetto tale attacco viene performato da un software scritto in linguaggio C che crea un pacchetto HTTP-GET molto semplice e sfrutta al massimo l'hardware della macchina (in particolare la banda a disposizione) per generare ed inviare quante più richieste possibili. Tramite questo software è possibile scegliere 3 parametri ovvero il numero di richieste che si vuole generare, il numero di connessioni concorrenti che si vogliono instaurare con il server e la scelta dell'header del pacchetto HTTP.

```c
// create the HTTP request buffer
int request_count = 0;
int response_count = 0;
int write_buffer_length = sprintf(write_buffer, "GET %s HTTP/1.1\r\nHost: %s\r\n", target.path, target.host);
i = 0;
while (headers[i]) {
    write_buffer_length += sprintf(write_buffer + write_buffer_length, "%s\r\n", headers[i]);
    i++;
}
write_buffer_length += sprintf(write_buffer + write_buffer_length, "\r\n");
```

```c
if ((events[i].events & EPOLLOUT && (max_requests == -1 || request_count < max_requests))) {
    // socket ready for writing
    if (client->pending == 0) {
        http_send_request(client, write_buffer, write_buffer_length);
        request_count++;
    }
}
```

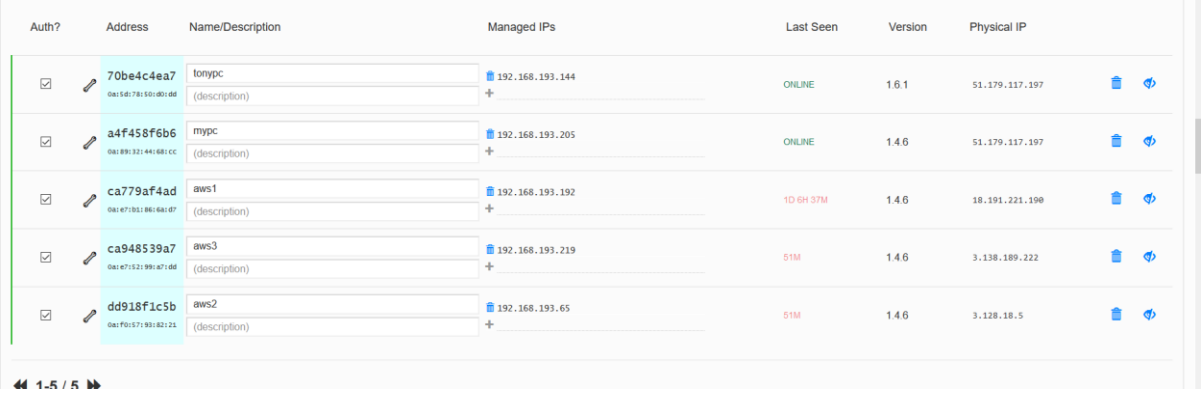
Quello che si ottiene dal punto di vista del pacchetto di rete, lanciando l'eseguibile compilato, è questo:



In questo esempio si apre una connessione tcp con il server e si inoltra la richiesta HTTP che verrà processata dal web server che risponderà a sua volta con tutto il necessario per visualizzare l'applicazione web. In questo caso la richiesta è unica ma grazie al binario **inundator** (compilato a partire dal programma httpFlooding) è possibile specificare il numero di richieste che si vuole generare e specificare quante connessioni concorrenti mantenere attive. Nelle varie prove effettuate la configurazione più adatta è stata quella di generare 40000 richieste gestite da 500 connessioni concorrenti per ogni istanza di EC2.

1.4 RETE PRIVATA ZEROTIER

Per far comunicare il tutto si è scelto di realizzare una rete privata dove svolgere le simulazioni. La scelta è stata fatta sia per una questione di semplicità in modo tale da non dover complicare il web server per essere raggiungibile dal mondo esterno, sia per una questione di sicurezza in quanto l'attacco **DDoS** è considerato **reato informatico** qualora comporti il danneggiamento o l'interruzione di un sistema informatico. Per questo motivo si è evitato di comprendere nella simulazione servizi di hosting e DNS per evitare qualsiasi tipo di problema. La soluzione utilizzata è stata di sfruttare il software **ZeroTierOne** che permette la creazione di una rete privata.



Tale software è stato scaricato e installato (in fase di configurazione) su tutti i dispositivi utilizzati per la realizzazione del progetto che più nello specifico sono:

- Web server ospitato su Windows (XAMPP)
- Istanze EC2 Ubuntu 20.04LTS
- Client benevolo Windows
- Client malevolo Kali linux

L'unico terminale che non fa parte della rete privata è proprio il Client malevolo in quanto grazie all'esecuzione remota di codice tramite ssh è in grado di far partire l'attacco al di fuori della rete privata. Tale pratica viene utilizzata spesso per simulare l'attacco ad una rete aziendale partendo dall'esterno della rete come se ci fosse una **backdoor**. Una volta installato e configurato ZeroTierOne sostanzialmente si va ad utilizzare un'interfaccia di rete virtuale su cui vengono instradate tutte le comunicazioni.

1.5 COMUNICAZIONI CLIENT MALEVOLO – ISTANZE EC2

La comunicazione con le istanze di EC2 ed il client malevolo che si trova al di fuori della rete privata avviene tramite una chiamata a procedura remota effettuata direttamente dal protocollo **ssh** e dal carattere **&** che in bash permette di eseguire in parallelo più comandi tra di loro:

//censored

Questo è lo script che viene eseguito sul client malevolo (kali linux) per simulare il controllo che si ha su dei terminali “infetti” (istanze EC2). Tale procedura poteva essere fatta in diversi modi (aprire una reverse shell sulle istanze EC2 simulando una falla di sicurezza) ma si è ritenuto questo metodo il più immediato ed efficace in quando è uno degli accessi che AWS stesso predilige per interagire con le proprie macchine.

2 RISULTATI DELL’ATTACCO

Dopo aver spiegato le principali componenti e come sono state implementate è necessario vedere i risultati del progetto. Avviando la simulazione possiamo notare come nella prima fase simuliamo che il client stia lavorando sul web server e quindi ci sia una comunicazione Client-Server ben instaurata.

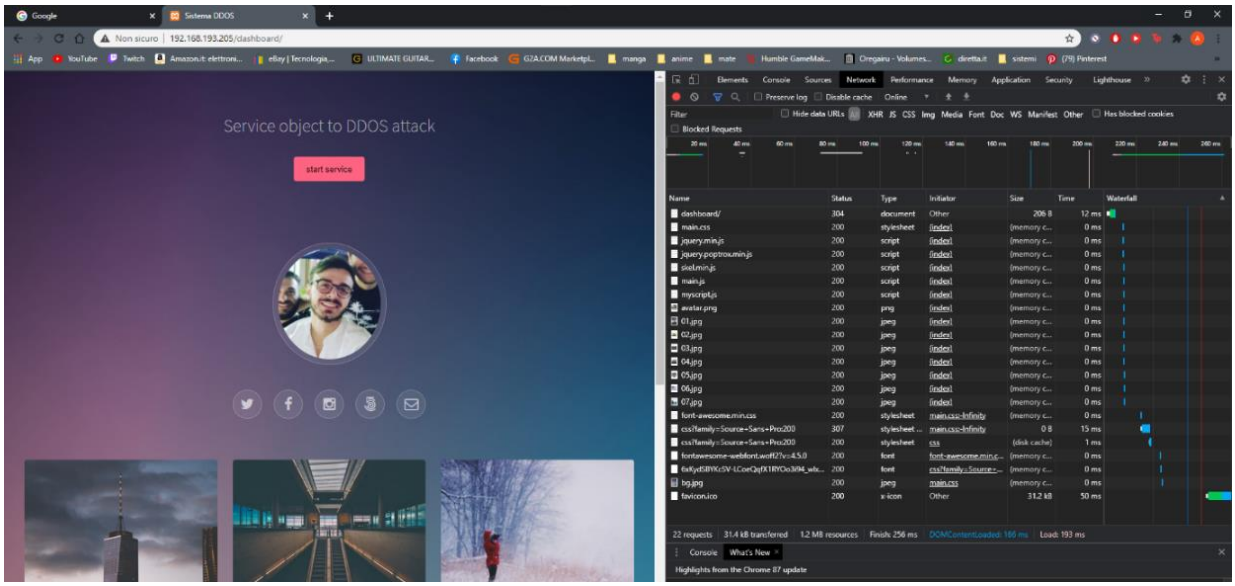


Figura 5: caricamento della pagina web

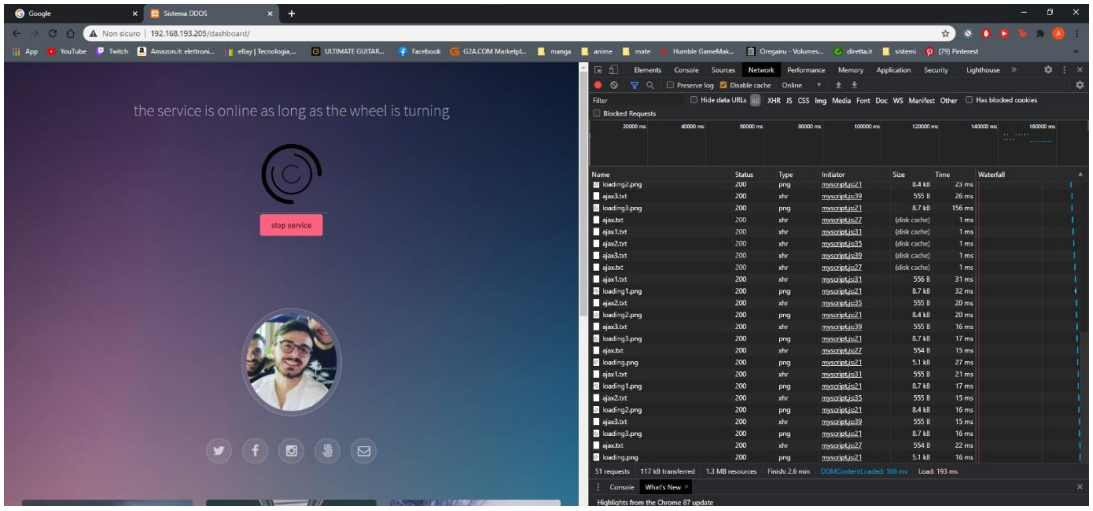


Figura 6: inizializzazione del servizio web

Successivamente sul client malevolo eseguiamo lo script che avvierà l'attacco DDoS.

```
lollo@kali:~/Desktop/ddos$ ./start.sh 40000 500 http://192.168.193.205/
lollo@kali:~/Desktop/ddos$ Requests sent: 40000
Responses received: 40000
Requests sent: 40000
Responses received: 40000
Requests sent: 40000
Responses received: 40000
^C
lollo@kali:~/Desktop/ddos$
[7] 0:bash*
```

E adesso notiamo come le richieste che venivano generate dal componente ajax incominciano a fallire bloccando di fatto il servizio. Inoltre l'utente benevolo, se provasse ad aggiornare la pagina, non riuscirebbe più ad ottenere l'accesso al sito web in quanto molto probabilmente il server non sarebbe più in grado di processare la sua richiesta.

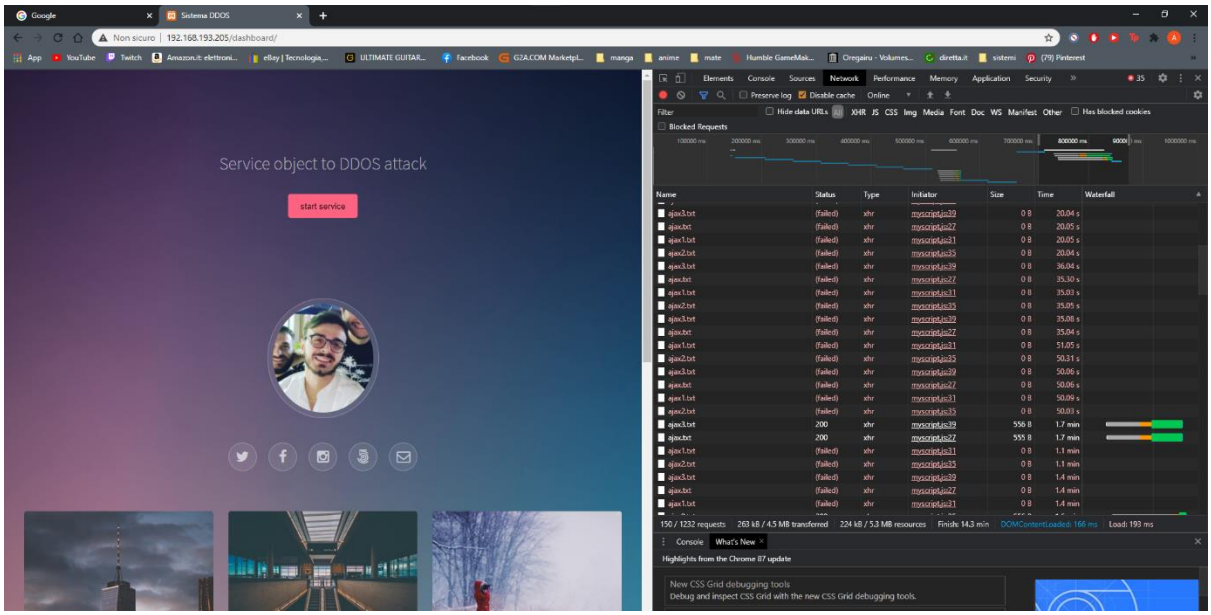


Figura 7: fallimento nell'erogazione del servizio

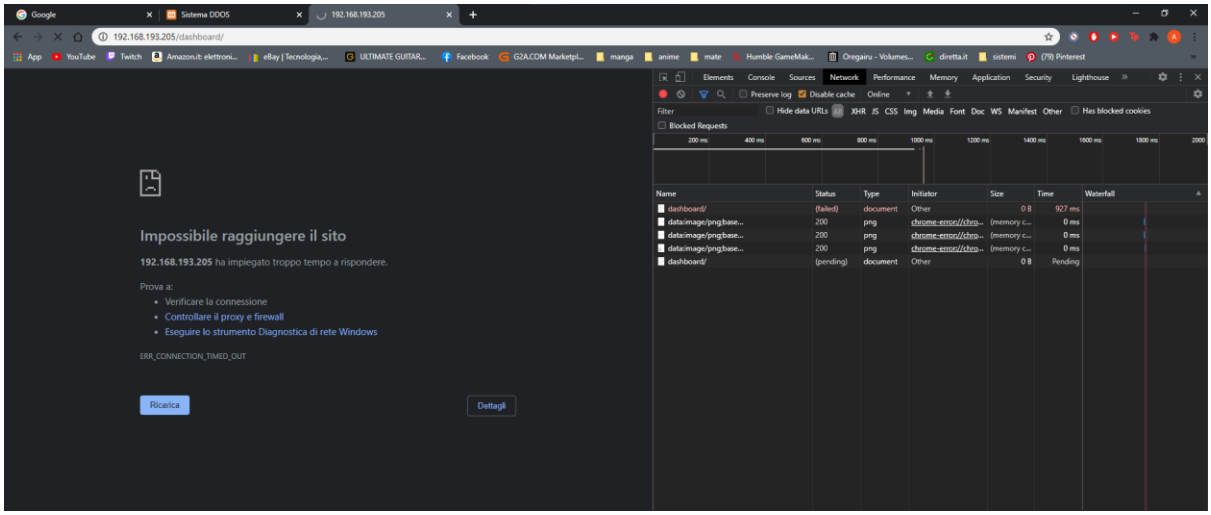


Figura 8: impossibilità di accedere alla pagina web

3 CONCLUSIONE

Con questo progetto si è voluto dimostrare quanto possa essere semplice al giorno d'oggi sovraccaricare un web server fatto in maniera poco curata e senza meccanismi di difesa ad attacchi DDoS. Tale approccio è reso estremamente semplice grazie ai servizi di **AWS** che in maniera gratuita (con i dovuti limiti) forniscono una forza computazionale e/o di rete potenzialmente fatale per tanti web server. L'immediatezza con cui è possibile reperire una tale forza computazionale al giorno d'oggi rende questa tipologia di attacchi spaventosamente efficace per chi abbia un minimo di esperienza nel campo dell'hacking e la possibilità in maniera così intuitiva di generare ed eliminare istanze di macchine virtuali rende anche il riconoscimento degli attaccanti molto più complesso per eventuali sistemi di sicurezza che applicano filtri in base all'IP. In conclusione, il cloud services ed in particolare i servizi di **IaaS**, opportunamente utilizzati, forniscono ulteriori possibilità di attacco ed elusione in ambito di sicurezza informatica oltre ad essere un valido metodo per ottenere risorse di calcolo in maniera semplice e veloce.