Sicurezza Informatica e Internet

Anno accademico 2015/2016

Relazione del progetto A.3

Build Your Own Botnet v. 2

Matteo Iacucci - 0211237 - matteoiacucci@gmail.com

Alessio Zannetti - 0221845 - alessio.zannetti@gmail.com

Indice

1 Introduzione

La seguente relazione riguarda la progettazione e lo sviluppo di un web service con un doppio comportamento a seconda del modo in cui ci si collega. Come noto nel protocollo HTTP ci sono vari campi che possono variare a seconda del client ed in base a tali campi si è sviluppato un servizio che rispondesse in maniera differente.
Il progetto prevede quindi l'interazione del web service con il client in modo canonico, presentando quindi una semplice pagina web se contattato canonicamente, come avviene , ad esempio, tramite un normale browser web, mentre, se contattato in modo alternativo, tramite una "codifica particolare", il server restituirà un contenuto diverso dalla pagina canonica, cioè una pagina web alternativa e un file contenente un insieme di coppie chiave valore. Le specifiche della modalità alternativa verranno spiegate ampiamente in seguito mostrando i modi con cui è possibile richiedere questo servizio alternativo al server.
In aggiunta alle richieste è stato implementata una pagina di gestione del server da parte dell'amministratore per la sua modalità alternativa, in modo da poter cambiare il file inviato e quindi avere un maggior controllo sul servizio offerto. Tale aggiunta è stata pensata per ottenere un service di tipo command and control. Per accedere a questa pagina di gestione anche in questo caso è stato adottato un metodo simile a quello per la richiesta della pagina alternativa con l'aggiunta della richiesta di una password.
In conclusione il server è stato messo in ascolto su due porte in modo tale da poter funzionare sia con protocollo non criptato HTTP che protocollo sicuro HTTPS in modo tale da evitare eavesdropping sulle informazioni inviate tra server e client.
Nei prossimi capitoli andremo a parlare delle botnet in generale per poi passare alla spiegazione del nostro web service, delle tecnologie usate per poterlo implementare e in fine mostreremo degli scenari di utilizzo del web service.

2 Background: BotNet

Un bot in generale è un programma che compie dei task in modo autonomo senza la necessità di interazione con un soggetto umano. I task che possono fare i bot sono di vario tipo e possono andare dal giocare in modo autonomo a poker a quello di cercare numeri primi.

Una BotNet è secondo la definizione data da Provos & Holz[1] "[...] network of compromised machines that can be remotely controlled by the attacker", cioè un network di macchine compromesse che possono essere controllate da remoto da un'entità malevola che coordina l'intero sistema. Quindi queste macchine compromesse eseguono a loro insaputa del codice che compie delle azione in modo autonomo. In altre parole le macchine diventano dei veri e propri bot, in breve quindi una botnet è un insieme di macchine compromesse, quindi di bot che vengono usate per scopi malvagi. Un altro componente fondamentale di una botnet è il botmaster che rappresenta la persona o il gruppo che controlla e gestisce la botnet.

Ci sono quattro aspetti fondamentali in una botnet:

- Il primo è che una botnet come suggerisce il nome è un network, quindi tutti i nodi interessati devono poter comunicare tra loro ed in particolare poter ricevere istruzioni dal master.
- Il secondo aspetto è che una macchina compromessa non sa di far parte di una botnet, la compromissione infatti avviene spesso tramite di un programma che di nascosto installa un componente che permette il collegamento della macchina alla botnet.
- La terza chiave di lettura di una botnet è stata data per scontanta precedentemente, ma è un aspetto molto importante, infatti una botnet deve essere in grado di essere comandata e controllata da remoto in modo da poter permettere alle persone che gestiscono la botnet di poter far eseguire i task che vengono inviati ai bot.
- Il quarto aspetto è che tipicamente una botnet è gestita da persone o gruppi che hanno intenti malvagi, che quindi utilizzano tali network per compiere azioni illegali quali attacchi DDos, smamming, sniffing e altre azioni simili.

[1] SANS – Institute. "BYOB: Build Your Own Botnet".

3 Sviluppo

In questo capitolo verrà illustrata come è stato sviluppato il web service e quali tenologie sono state usate.

3.1 Il web service

Il web service è stato pensato come un server web che rispondesse alle richieste dei client in modo tale da restituire il contenuto a seconda della codifica della richiesta. Per questo motivo è stato sviluppato con API RESTful che potessero garantire il perfetto funzionamento a seconda della richiesta. È stato usato il paradigma REST per implementare le funzionalità del nostro server poichè questo ci permetteva di gestire le richieste in modo semplice e poter rispondere in modo appropriato. Mediante questo paradigma quindi è stato possibile ricercare dentro le richieste HTTP i campi che ci interessavano per poter capire se il server fosse stato contattato in modo canonico o alternativo.
La codifica alternativa della richiesta è stato sviluppata in tre modi differenti, consigliati dalla traccia del progetto. Il primo metodo è quello di controllare se la richiesta alla pagina fosse inviata tramite protocollo HTTP post invece di una semplice GET. Il secondo metodo riguarda l'URL di richiesta, tramite l'aggiunta di alcuni valori i quali sono sottoposti ad un controllo di corrispondenza di parametri preimpostati, dopo il quale si ha accesso alla pagina alternativa. Ultimo metodo di contatto per la pagina alternativa è la modifica dell'User-Agent all'interno dei campi della richiesta HTTP, infatti impostando questo con un valore prestabilito, "BYOV v.2", il server risponde restituendo la pagina alternativa.
La gestione di queste modalità di contatto alternativo è stata semplificata dall'utilizzo del paradigma REST che ci ha quindi permesso una semplice e leggibile implementazione di tutto.

3.2 Il client ad hoc

Per quando riguarda l'estensione proposta, veniva richiesto di permettere al client che contattava il server con codifica alternativa di poter salvare in locale un file contenente delle coppie chiave-valore tutto in modo automatico. Per fare questo è stato pensato di aggiungere nella pagina non canonica uno script che automaticamente scaricasse sulla macchina che facesse la richiesta il file senza la necessità di un intervento di terzi. In questo

modo è stato in pratica implementato un client all'interno della pagina che potesse salvare in automatico tale file sulla macchina.

3.3 Estensioni

L'ultima estensione per il progetto introdotta da noi è, come accennato in precedenza, l'aggiunta di una pagina di amministrazione in modo tale da poter aggiornare il file descritto in precedenza in modo tale da consentire un controllo sul server da remoto tramite una semplice pagina. Anche in questo caso per poter accedere a tale pagina è necessario una codifica particolare della richiesta. Sono stati sviluppati due tipi di codifiche per poter accedere a questa pagina, una è quella mediante valori nell'URL e l'altro mediante la modifica dell'User-Agent nella richiesta HTTP.
Per concludere il server è stato dotato anche di una sicurezza chiave pubblica grazie alla creazione di un certificato SSL, in modo tale da poter contattare questo in modo sicuro, criptando il flusso di dati scambiato tra questo e i client che ne richiedono i servizi.

3.4 Tecnologie utilizzate

In questo paragrafo andremo a parlare delle tecnologie utilizzare per ci hanno permesso di realizzare il web service prima descritto.

3.4.1 Paradigma REST

REST, REpresentational State Transfer, rappresenta un tipo di architettura software per applicazioni Web adottato con ampia diffusione e diventato il modello predominante su Internet.
Questo paradigma architetturale pone al centro dell'attenzione non più i servizi, concetto prevalente nel cosiddetto Web 1.0, ma bensì le risorse: l'applicazione web viene considerata come un insieme di risorse, accessibili tramite il classico protocollo HTTP.
Nel nostro caso ciò che propone REST è creare un server capace di accettare tutti i metodi definiti dal protocollo HTTP, in particolar modo i quattro principali che permettono di effettuare le operazioni CRUD, ovvero: GET, PUT, POST e DELETE.
In particolare questo paradigma è stato sfruttato per realizzare un meccanismo che riconoscesse in maniera immediata il metodo con cui il servizio vine contattato, per poi rispondere in maniera adeguata; infatti è fondamentale distinguere il raggiungimento di talune risorse tramite il più tradizionale metodo GET tramite la quale viene restituita una comune pagina html, rispetto ad una più ricercata POST, tramite la quale viene presentata

una pagina alternativa con diverse features annesse. Tutto ciò consente un facile mascheramento di un possibile sito fraudolento se non controllato in maniera approfondita.

3.4.2 NodeJS

Node.js è un framework che usa un modello di networking non basato sui processi concorrenti ma un modello I/O event-driven: quindi Node richiede al sistema operativo di ricevere notifiche al verificarsi di derminati eventi, rimandando quindi l'elecuzione del codice relativo ad una chiamata di I/O a quando il sistema operativo avrà prelevato i dati dalla memoria di massa e li avrà resi disponibili all'applicazione, tramite una funzione di callback asincrona rispetto al flusso di esecuzione. Quindi appena l'applicazione in Node si avvia, dopo aver inizializzato le variabili, dichiarato le funzione d essersi registrato agli eventi, aspetta che accada uno di questi per eseguire la funzione di callback legata a tale evento. Un altro aspetto fondamentale del linguaggio Node.js è che le funzioni non sono bloccanti ciò ci permette di eseguire altre operazioni mentre si attende la risposta da un altra funzione. Questo aspetto è molto importante nell'utilizzo di Node per la scrittura di un servizio web based, in questo modo non c'è bisogno della definizione di più thread per il servizio di richieste multiple, semplificando di molto l'implementazione e rendendo il codice leggibile.

3.4.3 HTTP & HTTPS

L'HyperText Transfer Protocol (HTTP) è un protocollo a livello applicativo utilizzato per la trasmissione di informazioni tramite un meccanismo di richiesta/risposta tra client e server. Nel messaggio di richiesta sono definiti diversi campi tra cui, quelli a noi più utili, il tipo di richiesta (GET, POST) e lo User-Agent, identificativo del tipo di client utilizzato.
A questo protocollo è stato aggiunto un maggiore livello di sicurezza, diventando HTTPS, ovvero HTTP over Secure Socket Layer, inserendo la comunicazione all'interno di una connessione criptata attraverso un protocollo di crittografia asimmetrica realizzato dal Secure Sockets Layer (SSL) o meglio ancora dal Transport Layer Security (TLS).
Viene quindi creato un canale di comunicazione criptato attraverso uno scambio iniziale tra client e server di certificati, ovvero dei documenti elettronici che associano l'identità di un entità ad una chiave pubblica.
Il web service realizzato è in ascolto sia su una porta tramite la quale instaura una semplice comunicazione HTTP, che su un altra porta sulla quale viene instaurata una connessione sicura.

4 Implementazione

In questo capitolo andremo a descrivere l'implementazione del web service, scritto con il linguaggio Node.js e successivamente verrà descritta l'implementazione del client ad hoc per il salvataggio del file contenente le coppie chiave-valore sul computer che ne fa rchiesta.

4.1 Web Server

Come detto in precedenza il servr web è stato scritto in linguaggio NodeJS che ci ha permesso di implementarlo secondo il paradigma REST senza curarci di problemi riguardanti la concorrenza data la natura event-driven del linguaggio scelto.
Inizialmente vengono inizializzate tutte le variabili con i `require` necessari per le librerie usate. La libreria che si può dire più importante per la realizzazione del server è `express`. Questa libreria permette di realizzare con pochi passaggi di realizzare un server REST, che risponde in modo appropriato a seconda di come viene contattato il server.

```
var express = require('express');
var app = express();
app.get('/index.php', function(req, res){
    var varUrl = url.parse(req.url,true).query;
    //se contattato con parametri o con custom user agent viene restituita la pagina
htlm alternativa
    if(varUrl.var1=='1'&& varUrl.var2=='2' || req.headers['user-agent']==='BYOB v.2')
        res.sendFile(__dirname + '/indexPinoTroll.htm');
    else
    //altrimenti viene restituita la pagina canonica
        res.sendFile(__dirname + '/indexPino.htm');
});
```

Grazie a queste poche righe è stato possibile creare la prima parte del progetto. Se viene contattato il server richiedendo la pagina `index.php` il server restituisce la pagina a seconda della codifica della richiesta. Come si può notare se l'URL di richiesta contiene due parametri `var1` e `var2` settati rispettivamente a 1 e 2 o se il il campo User-Agent è uguale a `BYOB v.2` viene restituita la pagina alternativa, altrimenti la pagina canonica. Questo scenario avviene in caso di richiesta GET espresso dal metodo `app.get()`. L'altro metodo di richiesta con codifica alternativa prevedeva che la richiesta avvenisse tramite metodo POST ed anche in questo caso la semplicità di Node ci è venuta incontro infatti semplicemente cambiando il metodo `app.get()` in `app.post()` si risponde alla richiesta della pagina `index.php` se contattati tramite una POST.

```
app.post('/index.php', function(req, res){
    res.sendFile(__dirname + '/indexPinoTroll.htm');
})
```

Mediante l'utilizzo di questi metodi si è quindi potuto soddisfare le specifiche base andando a servire in modo corretto le diverse codifiche delle richieste inviate.
L'utilizzo della sola libreiria Express non permette in ogni caso il binding del servizio sulla macchina che esegue il codice, per rendere effettivamente il server esecutivo e in ascolto su una porta prefissata c'è bisogno dell'utilizzo della libreria http e la sua variante https per una connessione sicura.
Per quanto riguarda il binding del server senza sicurezza, dopo aver definito tutte le configurazioni di express per la risposta alla richieste entranti con il comando `http.createServer()` si crea il server http il quale mediante il metodo `listen()` viene messo in ascolto sulla porta stabilita in precedenza.

```
var httpPort = 8080;
var httpServer = http.createServer(app);
httpServer.listen(httpPort);
```

Molto simile è il meccanismo per creare il server con sicurezza SSL. In questo caso bisogna creare un oggetto json con due valori:

- key: contenende la chiave privata
- cert: contenente il certificato firmato

Creata tale variabile è possibile come nel caso precedente creare il server mediante il metodo `https.createServer()` per poi metterlo in ascolto su una porta scelta in precedenza.

```
var privateKey  = fs.readFileSync('sic/chiave.key', 'utf8');
var certificate = fs.readFileSync('sic/certificato.cert', 'utf8');
var credentials = {key: privateKey, cert: certificate};
var httpsPort = 8443;
var httpsServer = https.createServer(credentials, app);
httpsServer.listen(httpsPort);
```

Conclusi questi passaggi il server è funzionante e in ascolto su due porte differenti con due portocolli, uno senza cifratura dei dati in ingresso e in uscita un altro con cifratura mediante SSL.

4.1.1 Creazione dei certificati

I certificati per la creazione del server con cifratura sono stati generati tramite il software open source OpenSSL il quale permette di generare una chiave privata RSA e un certificato autofirmato. Questo certificato poiché non è stato firmato da nessuna autorità riconosciuta globalmente non verrà accettato inizialmente dal browser che ci consiglierà di non proseguire con la navigazione. Questo problema può essere risolto aggiungendo tale certificato a quelli trusted nel nostro browser.
Di seguito è riportato il comando utilizzato a tal fine:

```
$ openssl genrsa 1024 > chiave.key
$ chmod 400 chiave.key
$ openssl req -new -x509 -nodes -sha1 -days 365 -key chiave.key >
certificato.cert
```

4.2 Client ad Hoc

Per l'implementazione dell'estensione, cioè del salvataggio automatico di un file sul client che fa richiesta con codifica alternativa alla pagina si è aggiunta nel codice html della pagina una riga che permettesse al browser di scaricare in modo automatico il file.

```
<meta http-equiv="Refresh" content="0;URL=/file/byobFile">
```

Mediante l'utilizzo del tag `http-equiv="Refresh"` si reindirizza la pagina verso un nuovo URL che in questo caso è il file che deve essere salvato il locale, in questo modo il browser avvierà in modo automatico il download senza nessun input esterno.

4.3 Componenti aggiuntivi

Per completare le funzioni di command e control del server è stata aggiunta come detto in precedenza una pagina di amministrazione del server che serve per poter modificare il file consentente le chiave-valore.
Per poter accedere a tale pagina si è pensato un meccanismo simile a quello per la richiesta della pagina alternativa, infatti si può richiedere la pagina di login solo se si fa una richiesta

con parametri settati in un determinato modo o se i capo user-agent nella richiesta è BYOBAdmin.

```
//La pagina di login per accedere alla console di ammistrazione viene visualizzata solo
//se contattata tramite parametri o custom user agent
app.get('/login.html',function(req, res){
    var varUrl = url.parse(req.url,true).query;
    if(varUrl.var1=='3'&& varUrl.var2=='4' || req.headers['user-agent']==='BYOBAdmin')
        res.sendFile(__dirname + '/login.html');
    //se la richiesta non rispetta i vincoli viene ridiretta sull'index canonico
    else
        res.redirect('/index.php');
})
```

Similmente a prima si controlla quindi se nell'URL i parametri `var1` e `var2` sono settati rispettivamente a 3 o 4 o se user-agent è `BYOBAdmin`. Se la richiesta non dovesse rispettare questi vincoli si viene ridiretti sulla pagina canonica. In caso contrario viene restituita una pagina di login dove viene chiesta una password di accesso. Questa password verrà inviata al server nel campo body della richiesta successiva, mediante una richiesta POST.

```
//restituisce la pagina di console, solo se le credenziali immesse sono esatte.
//in particolare controlla se la password immessa coincide con quella salvata nel file
credential.json
app.post('/admin.html', checkAuth, function(req, res){
    var post = req.body;
    if (checkCredential(post.password)) {
        req.body.user_id = 'BYOBAdmin';
        res.sendFile(__dirname + '/admin.html');
    }
    else {
        res.send('Bad user/pass');
    }
 })
```

Viene quindi ricavata la password dal body della richiesta e grazie alla funzione `checkAuth` viene verificata con quella salvata in locale sul server. In caso di password errata verrà mostrato un messaggio di errore, in caso contrario verrà visualizzata la pagina di amministrazione.

Da questa pagina è possibile caricare il nuovo file che verrà salvato dal client quando contatterà il server in modalità alternativa. Per fare questo è stato aggiunta alla pagina un modulo per poter effettuare l'upload di un file locale che poi verrà inviato al server tramite una richiesta di tipo POST.

```
var storage = multer.diskStorage({
```

```
    destination: function (req, file, cb) {
        cb(null, './file')
    },
    filename: function (req, file, cb) {
        cb(null, 'byobFile')
    }
});
var upload = multer({ storage: storage });

//API REST per poter caricare il nuovo file da inviare in caso di richiesta alla pagina
alternativa
app.post('/upload', upload.single('uploaded'),function(req,res){
    res.send('upload completed')
})
```

Mediante al metodo REST `app.post("/upload")` viene chiamato il metodo `upload.single()` che salva in automatico il file nella destinazione prefissata mediante l'oggetto multer. In particolare il file verrà salvato nella cartella **"`./file`"** del server con il nome `byobFile`.

4.4 Logging

Un'altra feature aggiuntiva ed indispensabile in un servizio di questo genere è il salvataggio su di un semplice file di testo di tutte le operazioni effettuate sul server, in particolar modo la registrazione di tutti gli accessi secondo un particolare formato.
A tal proposito è stata utilizzata la libreria specifica per NodeJS **"`morgan`"** il cui riferimento è descritto dalla classica riga di codice.

```
var morgan = require('morgan');
```

Questa libreria permette di effettuare operazioni di logging in maniera semplice ed efficace tramite poche e semplici righe di codice nelle quali basta inserire il nome del file dove registrare le operazioni ed il tipo di formato che si vuole utilizzare.

```
app.use(morgan('combined',{
    stream: fs.createWriteStream('log.log', {'flags':'a'})
}))
```

Nel nostro caso si è scelto di utilizzare il formato preimpostato denominato "combined" che corrisponde a tipo di log offerto di default da Apache ed è sufficientemente ricco di informazioni in quanto permette di registrare i campi come di seguito riportato:

```
:remote-addr - :remote-user [:date[clf]] ":method :url HTTP/:http-version"
:status :res[content-length] ":referrer" ":user-agent"
```

Infine viene riportiamo un piccolo esempio di una portzione del file di log:

```
::ffff:10.220.61.194 - - [17/Jun/2016:10:28:08 +0000] "GET /index.php HTTP/1.1" 200
40118 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/51.0.2704.84 Safari/537.36"
::1 - - [24/Jun/2016:10:20:18 +0000] "GET /login.html HTTP/1.1" 200 343 "-" "BYOBAdmin"
::1 - - [04/Jul/2016:07:44:51 +0000] "GET /index.php?var1=1&var2=2 HTTP/1.1" 200 40231
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36"
```

5 Conclusioni

Utilizzando le tecnologie a disposizione come il linguaggio Node.Js e il paradigma REST si è potuto vedere come sia semplice l'implementazione di un server web che rispondesse in modo voluto a determinate richieste in entrata.
Il server da noi realizzato rappresenta un'ottima base per lo sviluppo di una botnet nella quale il server costituisce il nodo centrale, identificato dal botmaster, dal quale i bot ditribuiti nella rete possono ricevere i comandi per poter avviare un possibile attacco.
Un possibile sviluppo futuro sicuramente può riguardare il miglioramento della console di amministrazione, con l'aggiunta di ulteriori funzionalità per la gestione da parte del botmaster oltre alla gestione del singolo file che viene automaticamente scaricato dai client che ne fanno richiesta tramite la comunicazione alternativa.

6 Appendice

6.1 Installazione

Di seguito riportiamo una breve guida per l'installazione del framework principale e del packet manager annesso, in particolar modo per ambiente linux.
Per installare il framework Node.js è possibile scaricare l'ultima versione, per ciascuna piattaforma, dal sito ufficiale https://nodejs.org/download/release/ oppure tramite riga di comando:

```
$sudo apt-get install nodejs
```

In seguito sarà necessario installare il Nodejs Packet Manager (npm), indispensabile per l'importazione delle librerie esterne, tramite i seguenti comandi:

```
$curl https://www.npmjs.org/install.sh | sudo sh
$npm -v
```

Tramite npm sono state scaricate tutte le librerie utili per il nostro progetto. Tramite il comando npm --save sono stati salvati all'interno del file `package.json` tutti i nomi e le versioni di tali librerie. Sarà quindi sufficiente eseguire il comando:

```
$ npm install
```

Per poter installare tutte le librerie.
A questo punto si è pronti per eseguire il programma tramite l'istruzione:

```
$ node sicurezza.js
```

Una vota inizializzate tutte le variabili e le funzioni il programma mostrerà tale messaggio:

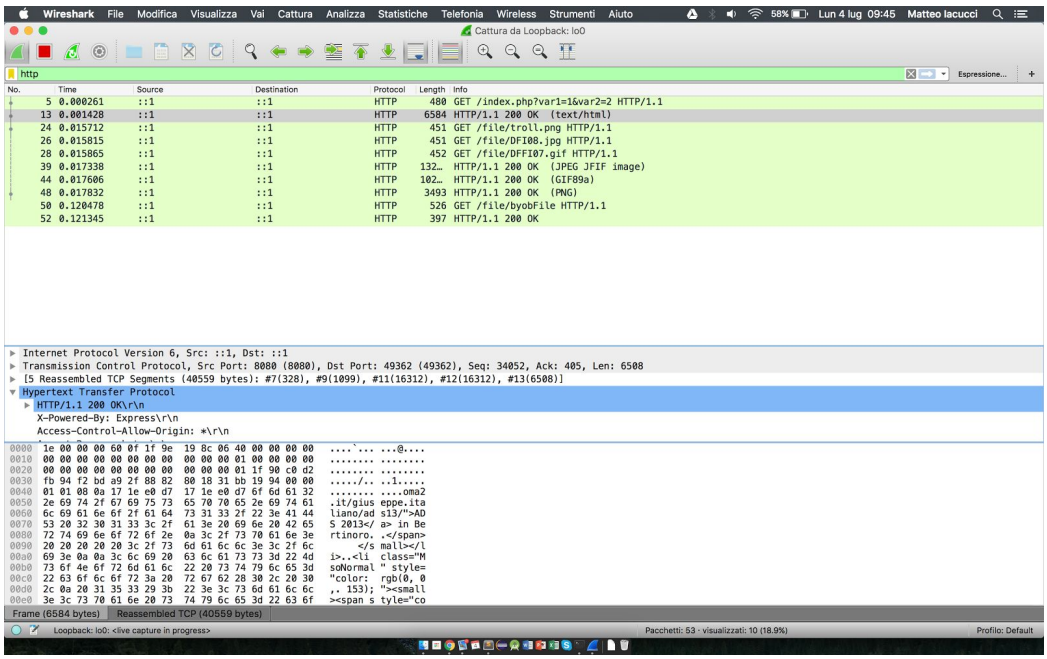
```
$ node sicurezza.js
```

```
Server is running
http Server on PORT      8080
https Server on PORT     8443
```

Ciò vuol dire che il server è pronto a ricevere connessioni sulle porte indicate con i meccanismi esposti in precedenza.

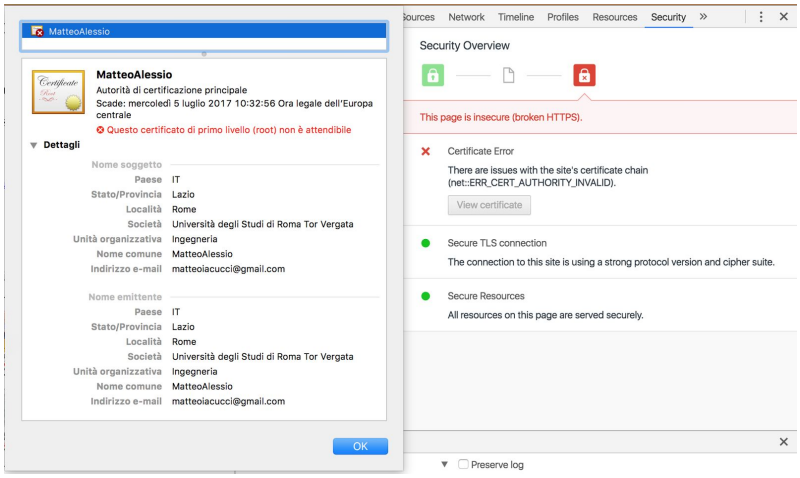
6.2 Scambio di messaggi

In questa seconda appendice mostreremo come i messaggi trasferiti tra il server e il client cambiano a seconda del protocollo utilizzato.



In questo primo screen si può vedere come lo scambio di messaggi tra server e client sia in chiaro e quindi ogni attaccante può vedere tutto il flusso di dati trasferito tra i due attori.

Nell’immagine successiva si può vedere invece come da browser web Chrome si possa osservare che la connessione è protetta con cifratura, ciò impedisce a qualunque attaccante di vedere il contenuto dei messaggi.



L’errore che viene visualizzato è dato dal fatto che il certificato è stato auto-firmato quindi non viene riconosciuto dal browser come sicuro ciò nonostante la connessione è criptata.