7 Den Burp-Proxy erweitern

Wenn Sie schon mal versucht haben, eine Webanwendung zu hacken, dann haben Sie sehr wahrscheinlich die Burp-Suite für Spider-, Proxy- oder andere Angriffe genutzt. Die neueren Versionen der Burp-Suite erlauben es, über sogenannte *Extensions* (also Erweiterungen) eigene Tools einzubinden.

Mit Python, Ruby oder reinem Java können Sie Panels und Automatisierungstechniken in die Burp-Suite integrieren. Wir wollen dieses Feature zu unserem Vorteil nutzen und Burp um einige praktische Tools erweitern, mit denen wir Angriffe durchführen und ausgedehntere Erkundungen betreiben können. Unsere erste Erweiterung nutzt einen vom Burp-Proxy abgefangenen HTTP-Request als Ausgangspunkt für einen Mutations-Fuzzer, der vom Burp Intruder ausgeführt werden kann. Die zweite Erweiterung nutzt Microsofts Bing-API, um uns alle virtuellen Hosts aufzuzeigen, die die gleiche IP-Adresse verwenden wie unsere Zielwebsite, sowie alle Subdomains, die für die Zieldomain erkannt werden.

Ich setze voraus, dass Sie bereits mit Burp gearbeitet haben und dass Sie wissen, wie man Requests mit dem Proxy-Tool abfängt und abgefangene Requests an den Burp Intruder sendet. Falls Sie eine Einführung brauchen, bietet PortSwigger Web Security (*http://www.portswigger.net/*) einen guten Einstieg.

Ich muss zugeben, dass bei meinen ersten Experimenten mit der Burp-Extender-API mehrere Versuche nötig waren, um zu verstehen, wie sie funktioniert. Als Python-Entwickler habe ich nur wenig Erfahrung mit der Java-Entwicklung und für mich war alles ein wenig verwirrend. Doch ich fand eine Reihe von Erweiterungen auf der Burp-Website, die mir zeigten, wie andere Leute Erweiterungen entwickelt haben, und nutzte diese, um zu verstehen, wie man eigenen Code implementiert. Ich werde einige Grundlagen zur Erweiterung der Funktionalität vermitteln, aber auch immer wieder zeigen, wie man die API-Dokumentation als Leitfaden für die Entwicklung eigener Erweiterungen nutzt.

7.1 Setup

Zuerst müssen Sie Burp von *http://www.portswigger.net/* herunterladen und installieren. So leid es mir tut, Sie benötigen eine moderne Java-Installation, für die

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

alle Betriebssysteme entweder Pakete oder Installer zur Verfügung stellen. Im nächsten Schritt laden Sie das Jython-Standalone-JAR (eine in Java geschriebene Python-Implementierung) herunter, die Burp später nutzen wird. Sie finden die JAR-Datei, zusammen mit dem restlichen Code aus diesem Buch, auf der Website von dpunkt (*http://www.dpunkt.de/mehr-python-hacking*) oder Sie besuchen die offizielle Site unter *http://www.jython.org/downloads.html* und wählen den Jython-2.7-Standalone-Installer. Lassen Sie sich durch den Namen nicht verwirren, es ist tatsächlich nur eine JAR-Datei. Speichern Sie die JAR-Datei an einer einfach zu merkenden Stelle, etwa dem Desktop.

Als Nächstes öffnen Sie ein Terminal und starten Burp wie folgt:

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

Damit wird Burp gestartet und es sollte dessen GUI voller wunderbarer Tabs erscheinen, wie in Abbildung 7–1 zu sehen ist.

Nun wollen wir dafür sorgen, das Burp unseren Jython-Interpreter nutzt. Klicken Sie auf den **Extender**-Tab und dann auf **Options**. Unter Python Environment geben Sie die Lage der Jython-JAR-Datei an (siehe Abb. 7–2).

Die restlichen Optionen können Sie unverändert lassen. Wir sind nun bereit, mit der Entwicklung unserer ersten Erweiterung zu beginnen. Los geht's!

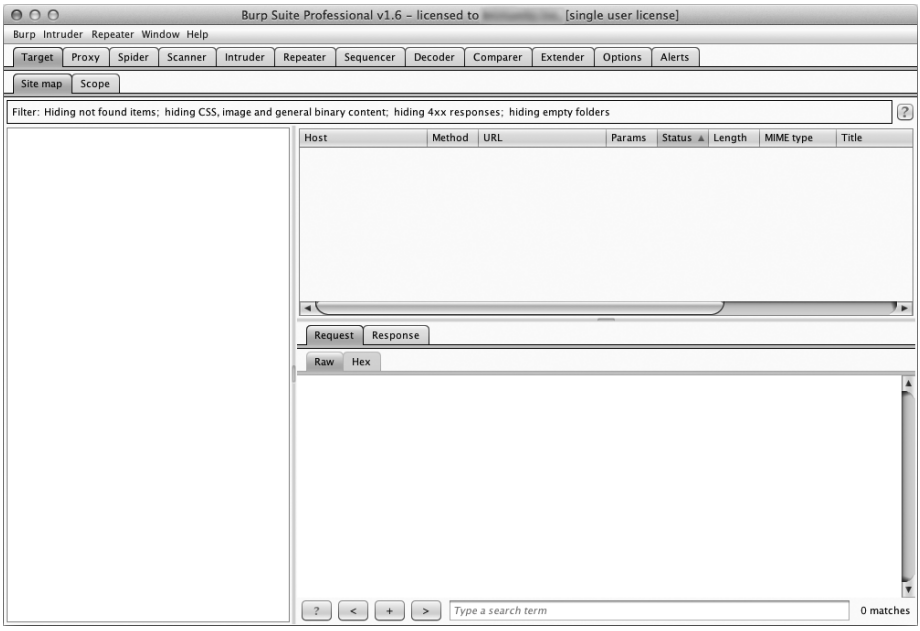

Abb. 7–1 *Burp-Suite-GUI erfolgreich geladen*

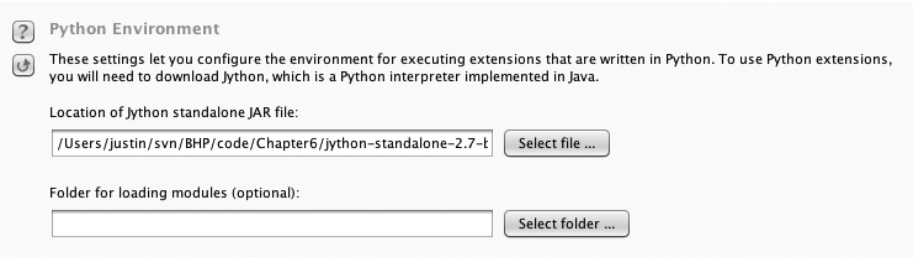

Abb. 7–2 *Pfad des Jython-Interpreters festlegen*

7.2 Burp Fuzzing

Irgendwann werden Sie vor dem Problem stehen, eine Webanwendung oder einen Webdienst zu attackieren, der die Nutzung traditioneller Tools für den Zugriff nicht erlaubt. Egal, ob ein innerhalb des HTTP-Traffics verpacktes binäres Protokoll oder komplexe JSON-Requests, es ist wichtig, dass Sie nach herkömmlichen Fehlern in Webanwendungen suchen können. Die Anwendung könnte zu viele Parameter nutzen oder in irgendeiner Weise verschlüsselt sein, sodass ein manueller Test viel zu lange dauern würde. Auch ich habe in vielen Fällen Standardtools genutzt, die nicht dafür ausgelegt sind, mit ungewöhnlichen Protokollen oder auch JSON umzugehen. An diesem Punkt ist es nützlich, wenn man mit Burp eine solide Basis für HTTP-Traffic (einschließlich Authentifizierungs-Cookies) aufbauen kann, während man den Body eines Requests an einen eigenen Fuzzer übergibt, der die Nutzdaten in jeder von Ihnen gewünschten Weise manipuliert. Wir beginnen unsere erste Burp-Extension mit der Entwicklung des einfachsten möglichen Webanwendung-Fuzzers, denn Sie dann ganz nach Wunsch erweitern können.

Burp besitzt eine Reihe von Tools, die Sie bei Tests von Webanwendungen nutzen können. Üblicherweise fangen Sie alle Requests mit dem Proxy ab und übergeben den Request dann an ein anderes Burp-Tool, wenn Sie etwas Interessantes entdecken. Ich nutze häufig das Repeater-Tool, mit dem ich den Web-Traffic wiedergeben und interessante Aspekte manuell verändern kann. Um den Angriff auf Query-Parameter etwas mehr zu automatisieren, senden Sie einen Request an das Intruder-Tool, das versucht, automatisch herauszufinden, welche Bereiche des Web-Traffics modifiziert werden sollen, und Ihnen dann die Verwendung unterschiedlicher Angriffe erlaubt, um Fehlermeldungen zu provozieren oder Sicherheitslücken zu offenbaren. Eine Burp-Extension kann auf vielerlei Weise mit den Tools der Burp-Suite interagieren. In unserem Fall werden wir die zusätzliche Funktionalität direkt in das Intruder-Tool integrieren.

Meinem natürlichen Instinkt folgend sehe ich mir zuerst die Burp-API-Dokumentation an, um herauszufinden, welche Burp-Klassen ich erweitern muss, um meine eigene Extension zu entwickeln. Sie können auf diese Dokumentation zugreifen, indem Sie zuerst den **Extender**-Tab und dann den **APIs**-Tab anklicken.

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

Das kann ein wenig beängstigend wirken, weil es sehr Java-lastig aussieht (und auch ist). Als Erstes bemerkt man, dass die Burp-Entwickler jede Klasse treffend benannt haben. Das heißt, es ist leicht herauszufinden, wo man anfangen muss. Da das Fuzzing der Web-Requests während eines Intruder-Angriffs erfolgen soll, fallen uns die `IIntruderPayloadGeneratorFactory`- und `IIntruderPayloadGenerator`-Klassen ins Auge. Sehen wir uns an, was die Dokumentation über die `IIntruderPayloadGeneratorFactory`-Klasse zu sagen hat:

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

Am Anfang der Dokumentation ❶ wird uns mitgeteilt, dass die Erweiterung korrekt bei Burp registriert werden muss. Wir werden sowohl die Haupt-Burp-Klasse als auch die `IIntruderPayloadGeneratorFactory`-Klasse erweitern. Als Nächstes stellen wir fest, dass Burp zwei Funktionen in unserer Hauptklasse erwartet. Die Funktion `getGeneratorName` ❷ wird von Burp aufgerufen, um den Namen unserer Erweiterung abzurufen. Dabei wird von uns erwartet, dass wir einen String zurückgeben. Die Funktion `createNewInstance` ❸ verlangt von uns die Rückgabe

einer Instanz des `IIntruderPayloadGenerators`, der zweiten Klasse, die wir entwickeln müssen.

Lassen Sie uns nun den Python-Code implementieren, der diese Bedingungen erfüllt. Danach sehen wir uns an, wie die Klasse `IIntruderPayloadGenerator` hinzugefügt wird. Öffnen Sie dazu eine neue Python-Datei namens *bhp_fuzzer.py* und geben Sie den folgenden Code ein:

```
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)
```

Das ist also unser einfaches Grundgerüst, um die ersten Anforderungen an unsere Erweiterung zu erfüllen. Zuerst müssen wir die `IBurpExtender`-Klasse importieren ❶. Das ist eine Forderung an jede von uns entwickelte Erweiterung. Dem folgt der Import der notwendigen Klassen für die Entwicklung eines Intruder-Nutzdatengenerators. Als Nächstes definieren wir unsere `BurpExtender`-Klasse ❷, die die `IBurpExtender`- und `IIntruderPayloadGeneratorFactory`-Klassen erweitert. Wir nutzen dann die Funktion `registerIntruderPayloadGeneratorFactory` ❸, um unsere Klasse zu registrieren, damit das Intruder-Tool weiß, dass wir Nutzdaten generieren können. Anschließend implementieren wir die Funktion `getGeneratorName` ❹, die einfach den Namen unseres Nutzdatengenerators zurückgibt. Unser letzter Schritt ist die Funktion `createNewInstance` ❺, die die Angriffsparameter empfängt und eine Instanz der `IIntruderPayloadGenerator`-Klasse zurückgibt, die wir `BHPFuzzer` genannt haben.

Werfen wir einen Blick auf die Dokumentation der `IIntruderPayloadGenerator`-Klasse, damit wir wissen, wie diese zu implementieren ist.

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

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

O.K.! Wir müssen also die Basisklasse implementieren und diese muss drei Funktionen bereitstellen. Die erste Funktion, `hasMorePayloads` ❶, entscheidet einfach, ob weitere mutierte Requests an den Burp Intruder zurückgegeben werden sollen. Wir handhaben das mit einem einfachen Zähler. Hat dieser Zähler ein von uns festgesetztes Maximum erreicht, liefern wir `False` zurück, sodass keine weiteren Mutationen generiert werden. Die Funktion `getNextPayload` ❷ empfängt die Originalnutzdaten des durch Sie abgefangenen HTTP-Requests. Haben Sie mehrere Nutzdatenbereiche im HTTP-Request gewählt, erhalten Sie hingegen nur die Bytes, die Sie mutieren wollen (mehr dazu später). Diese Funktion erlaubt das

Fuzzing der ursprünglichen Testdaten, die dann an Burp zurückgegeben werden, der diese neuen Werte anschließend sendet. Die letzte Funktion, `reset` ❸, erlaubt uns nach der Generierung einer bekannten Anzahl mutierter Requests – sagen wir fünf – für jede im Intruder-Tab festgelegte Nutzdatenposition eine Iteration über diese fünf mutierten Werte.

Unser Fuzzer ist gar nicht so »fussy« (dtsch.: »wählerisch«) und führt für jeden HTTP-Request nur ein zufälliges Fuzzing durch. Sehen wir uns nun an, wie das in Python implementiert wird. Fügen Sie den folgenden Code an das Ende von *bhp_fuzzer.py* an:

```
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):

          # In String konvertieren
❺         payload = "".join(chr(x) for x in current_payload)

          # Unseren einfachen Mutator für den POST aufrufen
❻         payload = self.mutate_payload(payload)

          # Anzahl der Fuzzing-Versuche erhöhen
❼         self.num_iterations += 1

          return payload

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

Wir beginnen mit der Definition unserer `BHPFuzzer`-Klasse ❶, die die Klasse `IIntruderPayloadGenerator` erweitert. Wir definieren die benötigten Klassenvariablen und fügen die Variablen `max_payloads` ❷ und `num_iterations` hinzu, damit wir nachverfolgen können, wann Burp mit dem Fuzzing fertig ist. Natürlich können Sie die Erweiterung immerzu durchlaufen lassen, wenn Sie das wünschen, doch zum Testen belassen wir alles erst einmal so. Als Nächstes implementieren wir die Funktion `hasMorePayloads` ❸, die einfach überprüft, ob wir die maximale Anzahl

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

unserer Fuzzing-Iterationen erreicht haben. Sie können die Erweiterung fortwährend durchlaufen lassen, indem Sie diese Funktion immer `True` zurückgeben lassen. Die Funktion `getNextPayload` ❹ erhält die ursprünglichen HTTP-Nutzdaten und führt das eigentliche Fuzzing durch. Die Variable `current_payload` enthält ein Byte-Array, das wir in einen String umwandeln ❺ und an unsere Fuzzing-Funktion `mutate_payload` ❻ weitergeben. Nun inkrementieren wir die Variable `num_iterations` ❼ und liefern die mutierten Nutzdaten zurück. Unsere letzte Funktion ist `reset`, die einfach zurückkehrt, ohne etwas getan zu haben.

Jetzt wollen wir noch die einfachste Fuzzing-Funktion überhaupt integrieren, die Sie ganz nach Herzenslust anpassen können. Da diese Funktion die aktuellen Nutzdaten kennt, können Sie bei kniffeligen Protokollen, die eine Sonderbehandlung verlangen (etwa eine CRC-Prüfsumme zu Beginn der Nutzdaten oder eines Längenfelds), alle nötigen Berechnungen innerhalb der Funktion vornehmen, was die Funktion besonders flexibel macht. Fügen Sie den folgenden Code in *bhp_fuzzer.py* ein und stellen Sie sicher, dass die Funktion `mutate_payload` in unserer `BHPFuzzer`-Klasse bekannt ist:

```
def mutate_payload(self,original_payload):
    # Einfachen Mutator auswählen oder ein externes Skript aufrufen
    picker = random.randint(1,3)

    # Zufälligen Offset für die Mutation in den Nutzdaten auswählen
    offset  = random.randint(0,len(original_payload)-1)
    payload = original_payload[:offset]

    # SQL-Injection an zufälligem Offset einfügen
    if picker == 1:
        payload += "'"

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

    # Originalnutzdaten zufällig oft einfügen
    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]

    # Restliche Teile der Nutzdaten anhängen
    payload += original_payload[offset:]

    return payload
```

Dieser einfache Fuzzer ist mehr oder weniger selbsterklärend. Wir wählen zufällig einen von drei Mutatoren aus: einen einfachen SQL-Injection-Test mit einem einzelnen Anführungszeichen, einen XSS-Versuch oder einen Mutator, der einen Bereich der Originalnutzdaten auswählt und zufällig oft wiederholt. Wir besitzen nun eine funktionsfähige Burp-Intruder-Erweiterung. Mal sehen, wie wir sie geladen bekommen.

Die Probe aufs Exempel

Zuerst müssen wir unsere Erweiterung laden und sicherstellen, dass keine Fehler auftreten. Klicken Sie den **Extender**-Tab an und dann den **Add**-Button. Ein Dialog erscheint, in dem Sie Burp den Pfad auf unseren Fuzzer bekanntgeben können. Verwenden Sie dabei die gleichen Optionen wie in Abbildung 7–3.

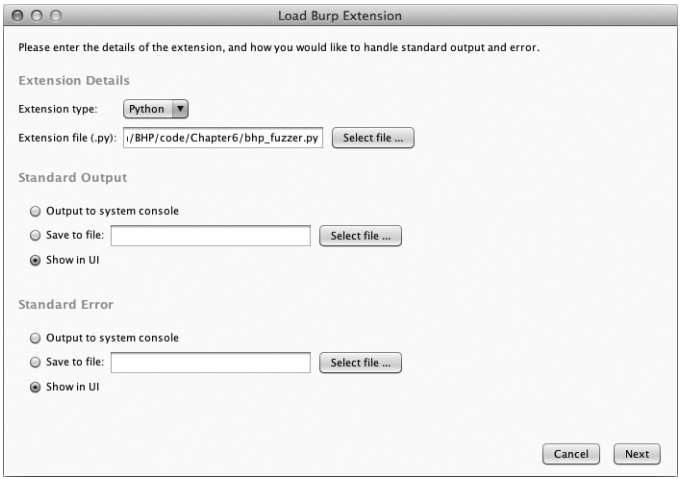

Abb. 7–3 *Einstellungen für das Laden unserer Erweiterung in Burp*

Klicken Sie dann auf **Next** und Burp beginnt mit dem Laden unserer Erweiterung. Geht alles gut, zeigt Burp an, dass die Erweiterung erfolgreich geladen wurde. Gibt es Fehler, klicken Sie den **Errors**-Tab an, korrigieren mögliche Schreibfehler und klicken dann auf **Close**. Die Extender-Seite sollte aussehen wie in Abbildung 7–4.

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

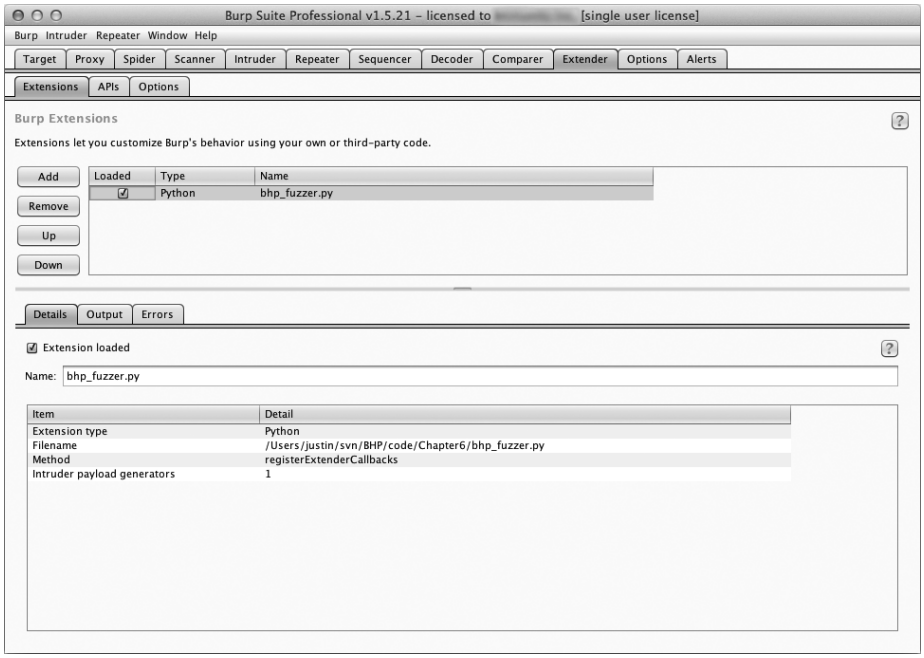

Abb. 7–4 *Burp Extender mit geladener Erweiterung*

Sie können sehen, dass unsere Erweiterung geladen wurde und Burp erkannt hat, dass ein Intruder-Nutzdatengenerator registriert wurde. Wir können unsere Erweiterung nun in einem realen Angriff einsetzen. Stellen Sie sicher, dass Ihr Webbrowser den Burp-Proxy als Localhost-Proxy an Port 8080 nutzt. Wir greifen nun die gleiche Acunetix-Webanwendung an, die wir schon aus Kapitel 6 kennen. Gehen Sie einfach auf die folgende Seite:

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

Ich habe beispielhaft das kleine Suchfeld auf der Seite genutzt, um nach dem String »test« zu suchen. Abbildung 7–5 zeigt diesen Request im HTTP-History-Tab des Proxy-Tabs. Ein Rechtsklick auf den Request sorgt dafür, dass er an den Intruder gesendet wird.

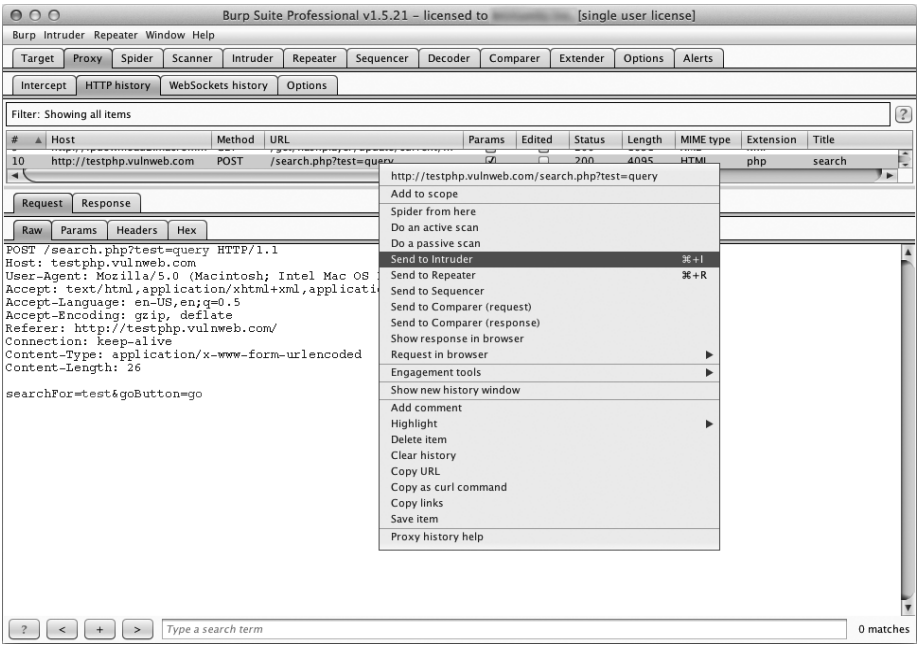

Abb. 7–5 *Einen HTTP-Request an Intruder senden*

Nun wechsele ich in den **Intruder**-Tab und klicke den **Positions**-Tab an. Es erscheint eine Seite, auf der jeder Query-Parameter hervorgehoben wird. Burp identifiziert hier die Elemente, an denen das Fuzzing erfolgen sollte. Sie könnten nun die Eingrenzung der Nutzdaten verändern oder die gesamten Nutzdaten mutieren, doch in unserem Beispiel wollen wir Burp entscheiden lassen, wo das Fuzzing erfolgen soll. In Abbildung 7–6 wird deutlich, wie das Nutzdaten-Highlighting funktioniert.

Klicken Sie jetzt den **Payloads**-Tab an. In dieser Maske wählen Sie im Dropdown-Menü **Payload type** den Eintrag **Extension-generated**. Im Abschnitt Payload Options klicken Sie den Button **Select generator...** an und wählen dort **BHP Payload Generator** aus. Die Payload-Maske sollte so aussehen wie in Abbildung 7–7.

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

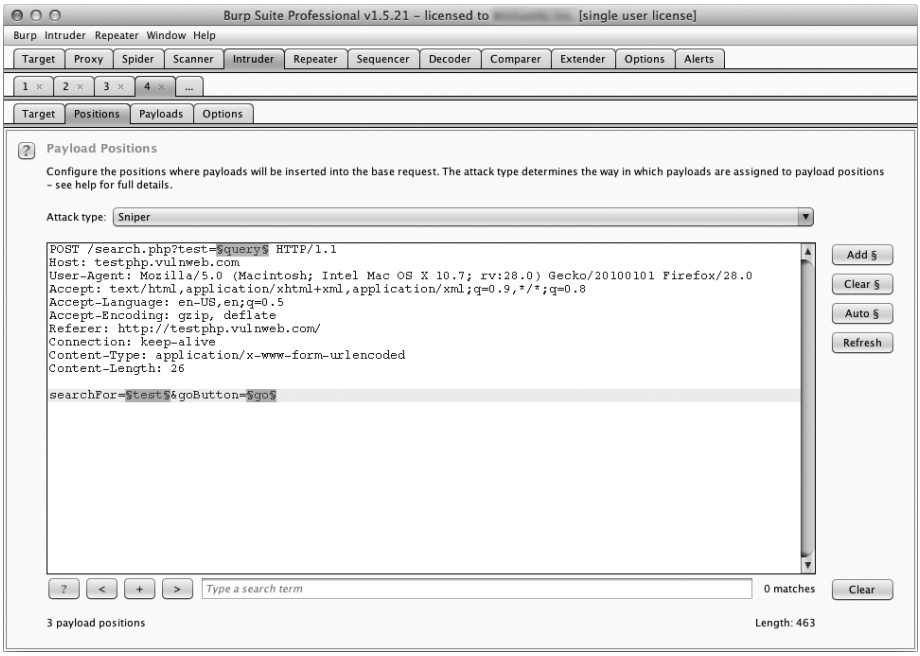

Abb. 7–6 *Burp Intruder mit hervorgehobenen Payload-Parametern*

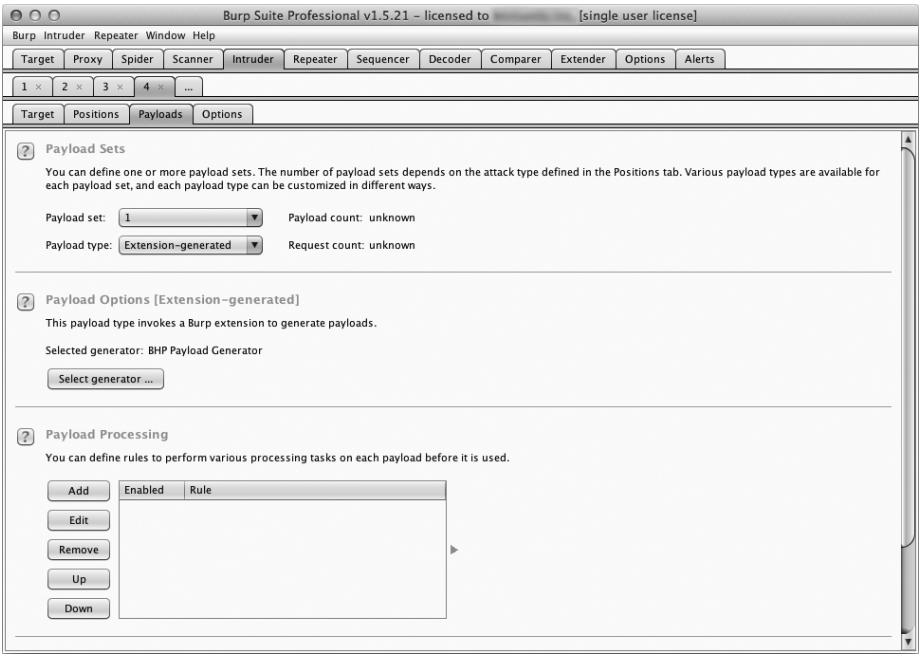

Abb. 7–7 *Unsere Fuzzing-Erweiterung als Nutzdatengenerator verwenden*

Wir sind jetzt so weit, dass wir unsere Requests senden können. In der oberen Burp-Menüleiste klicken Sie auf **Intruder** und wählen dann **Start Attack**. Das Senden mutierter Requests wird jetzt gestartet und Sie können die Ergebnisse schnell durchgehen. Die Ergebnisse meines Fuzzer-Laufs sehen Sie in Abbildung 7–8.

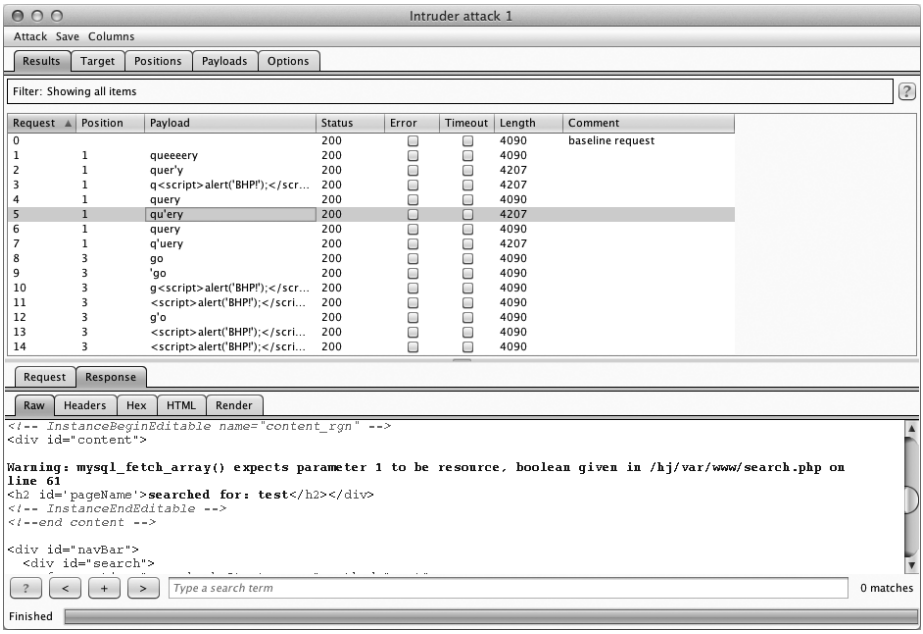

Abb. 7–8 *Unser Fuzzer während eines Intruder-Angriffs*

Wie Sie an der Warnung in Zeile 61 der Response erkennen, haben wir in Request 5 eine SQL-Injection-Lücke entdeckt.

Unser Fuzzer dient natürlich nur der Demonstration, doch Sie werden überrascht sein, wie effizient er Webanwendungen dazu bringt, Fehler zu generieren, Anwendungspfade zu offenbaren oder sich in einer Weise zu verhalten, die andere Scanner nicht mitbekommen. Es ist dabei wichtig, dass Sie verstehen, wie wir unsere Erweiterung mit Intruder-Angriffen in Einklang gebracht haben. Wir wollen uns nun einer Erweiterung zuwenden, die uns dabei hilft, einen Webserver etwas intensiver auszuspähen.

7.3 Bing für Burp

Bei Webservern ist es nicht ungewöhnlich, dass auf einem einzelnen Rechner mehrere Webanwendungen laufen, von denen Sie nichts wissen. Natürlich wollen wir alle Hostnamen kennen, unter denen ein Webserver zugänglich ist, da sie uns möglicherweise einen einfacheren Weg bieten, an eine Shell zu gelangen. Nicht selten findet man eine unsichere Webanwendung oder sogar Entwicklungsressourcen auf dem gleichen Zielrechner. Microsofts Suchmaschine Bing bietet die Möglich-

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

keit, über den Suchmodifikator »IP« eine Suche für alle Websites durchzuführen, die sie für eine einzelne IP-Adresse findet. Bing nennt Ihnen über den »domain«-Modifikator auch alle Subdomains für einen gegebenen Domainnamen.

Natürlich könnten wir einen Scraper nutzen, um diese Queries an Bing zu senden, und den HTML-Code der Ergebnisse verarbeiten, doch das wären schlechte Manieren (und würde die Nutzungsbedingungen der meisten Suchmaschinen verletzen). Um also unnötigem Ärger aus dem Weg zu gehen, verwenden wir die Bing-API[1] zum automatisierten Senden der Queries und verarbeiten die Ergebnisse dann selbst. Wir werden (außer einem Kontextmenü) keine Ergänzung der Burp-GUI vornehmen, sondern einfach das Ergebnis ausgeben, wenn wir eine Query ausführen. Jede erkannte URL wird automatisch in Burps Zielliste übernommen. Da Sie bereits wissen, wie man die Burp-API-Dokumentation liest und auf Python überträgt, wenden wir uns direkt dem Code zu.

Öffnen Sie *bhp_bing.py* und geben Sie den folgenden Code ein:

```
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 = "IHRSCHLÜSSEL"

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

        # Unsere Erweiterung einrichten
        callbacks.setExtensionName("BHP Bing")
❸       callbacks.registerContextMenuFactory(self)

        return

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

1. Unter *http://www.bing.com/dev/en-us/dev-center/* erhalten Sie einen eigenen, freien Bing-API-Schlüssel.

Das ist der erste Teil unserer Bing-Erweiterung. Tragen Sie Ihren Bing-API-Schlüssel an der entsprechenden Stelle ein ❶; ca. 2500 Suchanfragen monatlich sind kostenlos. Wir beginnen mit der Definition unserer `BurpExtender`-Klasse ❷, die das Standard-`IBurpExtender`-Interface implementiert, sowie `IContextMenuFactory`, über die wir ein Kontextmenü anbieten können, wenn der Benutzer in Burp einen Request mit der rechten Maustaste anklickt. Wir registrieren unseren Menü-Handler ❸, um zu ermitteln, welche Site der Benutzer angeklickt hat, sodass wir entsprechende Bing-Queries generieren können. Dann richten wir die Funktion `createMenuItem` ein, die ein `IContextMenuInvocation`-Objekt empfängt, mit dem wir den gewählten HTTP-Request bestimmen können. Der letzte Schritt besteht darin, unseren Menüeintrag zu erstellen und die `bing_menu`-Funktion das Klick-Event ❹ verarbeiten zu lassen. Jetzt wollen wir die eigentliche Bing-Query vornehmen, die Ergebnisse ausgeben und die erkannten virtuellen Hosts in Burps Zielliste aufnehmen.

```
    def bing_menu(self,event):

        # Was hat der Benutzer angeklickt
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):

        # Auf IP-Adresse oder Hostnamen prüfen
        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)
```

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

Unsere `bing_menu`-Funktion wird aufgerufen, wenn der Benutzer das von uns definierte Kontextmenü-Element anklickt. Wir rufen alle ausgewählten HTTP-Requests ab ❶ und dann den Hostteil jedes Requests, den wir dann zur weiteren Verarbeitung an unsere `bing_search`-Funktion übergeben. Die Funktion `bing_search` bestimmt zuerst, ob wir eine IP-Adresse oder einen Hostnamen übergeben haben ❷. Wir fragen dann bei Bing nach allen virtuellen Hosts mit der gleichen IP-Adresse ❸ wie der Host, der im angeklickten HTTP-Request enthalten war. Wurde eine Domain an unsere Erweiterung übergeben, führen wir noch eine zweite Suche durch ❹, die alle Subdomains umfasst, die Bing indexiert hat. Jetzt wollen wir die eigentliche Funktion implementieren, die Burps HTTP-API nutzt, um den Request an Bing zu senden und die zurückgegebenen Ergebnisse zu verarbeiten. Fügen Sie den folgenden Code hinzu und achten Sie darauf, dass Sie in der `BurpExtender`-Klasse sind, sonst kommt es zu Fehlern.

```
    def bing_query(self,bing_query_string):

        print "Performing Bing search: %s" % bing_query_string

        # Query codieren
        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
```

O.K.! Burps HTTP-API verlangt von uns, dass wir den gesamten HTTP-Request zuerst als String zusammenbauen, bevor er gesendet werden kann. Wie Sie sehen können, müssen wir für den API-Aufruf unseren Bing-API-Schlüssel base64-codieren ❶ und die HTTP-Basic-Authentifizierung nutzen. Dann senden wir unseren HTTP-Request ❷ an die Microsoft-Server. Wir erhalten die vollständige Antwort inklusive der Header zurück, weshalb wir die Header aussortieren ❸ und den Rest an unseren JSON-Parser übergeben ❹. Zu jedem Ergebnis geben wir einige Informationen über die entdeckte Website aus ❺ und wenn die entdeckte Site nicht in Burps Zielliste vorliegt ❻, fügen wir sie automatisch hinzu. Das ist ein schönes Beispiel dafür, wie man mit der Jython-API und reinem Python eine Burp-Erweiterung entwickelt, die beim Angriff auf ein bestimmtes Ziel zusätzliche Aufklärungsarbeiten übernimmt. Sehen wir uns das in Aktion an.

Die Probe aufs Exempel

Wir nutzen die gleiche Prozedur wie bei unserer Fuzzing-Erweiterung, um unsere Bing-Suche zum Laufen zu bekommen. Sobald sie geladen ist, gehen wir im Browser zu *http://testphp.vulnweb.com/* und wählen den gerade gesendeten GET-Request mit einem Rechtsklick aus. Wenn die Erweiterung korrekt geladen wurde, erscheint die Menüoption **Send to Bing** wie in Abbildung 7–9 zu sehen.

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

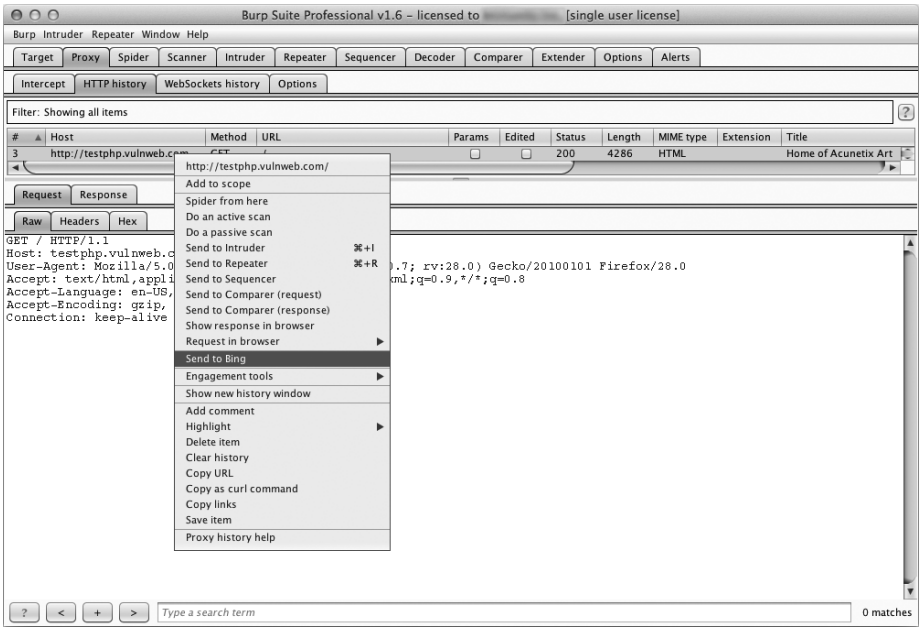

Abb. 7–9 *Neue Menüoption zeigt unsere Erweiterung.*

Klicken Sie diese Menüoption an, dann erscheinen (je nach gewählter Ausgabeoption) die Bing-Ergebnisse wie in Abbildung 7–10 zu sehen.

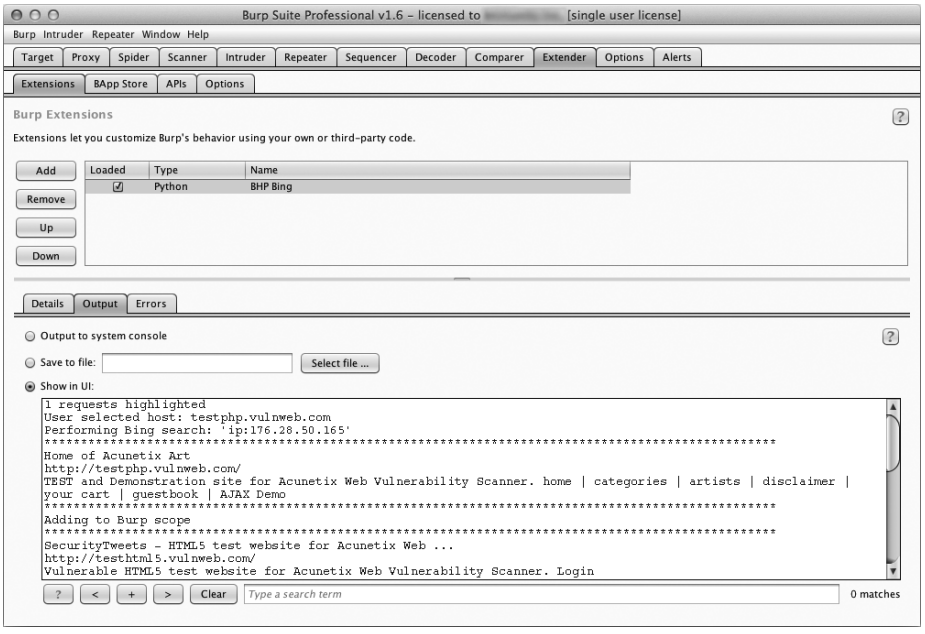

Abb. 7–10 *Die Erweiterung liefert die Ergebnisse unserer Bing-API-Suche.*

Klicken Sie nun den **Target**-Tab in Burp an und wählen Sie **Scope**, dann sehen die automatisch zur Zielliste neu hinzugefügten Elemente (Abbildung 7–11). Diese Zielliste beschränkt Aktivitäten wie Angriffe, Spidering und Scans auf die definierten Hosts.

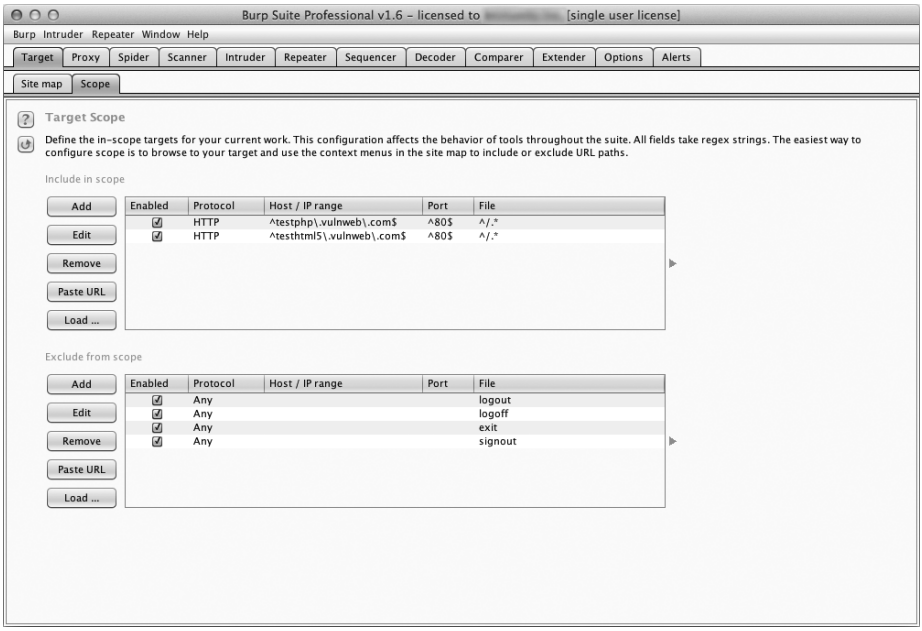

Abb. 7–11 *Erkannte Hosts erscheinen automatisch in Burps Zielliste.*

7.4 Website-Inhalte in Passwort-Gold verwandeln

Das Thema Sicherheit beschränkt sich – traurig aber wahr – häufig auf ein Wort: Benutzerpasswörter. Wenn es um Webanwendungen geht, insbesondere um selbstentwickelte, werden die Dinge noch schlimmer, weil häufig keine Account-Sperren implementiert werden oder starke Passwörter nicht erzwungen werden. In solchen Fällen kann eine Passwortattacke wie im letzten Kapitel die Eintrittskarte zu dieser Website sein.

Der Trick beim Erraten von Online-Passwörtern ist die Wahl der richtigen Wortliste. Wenn Sie es eilig haben, können Sie keine 10 Millionen Passwörter durchprobieren, weshalb Sie in der Lage sein müssen, eine für die fragliche Site geeignete Wortliste zu erzeugen. Natürlich gibt es in der Kali Linux-Distribution Skripte, die eine Website absuchen und aus deren Inhalt Wortlisten erzeugen. Doch wenn Sie bereits den Burp Spider nutzen, um eine Site zu verarbeiten, warum sollten Sie dann zusätzlichen Traffic erzeugen, nur um eine Wortliste zu generieren? Darüber hinaus verwenden diese Skripte üblicherweise Unmengen an Kommandozeilenparametern, die man sich merken muss. Wenn es Ihnen so geht

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

wie mir, dann kennen Sie bereits mehr als genug Kommandozeilenargumente, um Ihre Freunde zu beeindrucken, also überlassen wir doch lieber Burp die eigentliche Arbeit.

Öffnen Sie *bhp_wordlist.py* und geben Sie den folgenden Code ein.

```
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()

        # Wir beginnen mit etwas weit Verbreitetem
❹       self.wordlist   = set(["password"])

        # Wir richten unsere Erweiterung ein
        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
```

Der Code dieses Listing sollte Ihnen mittlerweile vertraut sein. Wir beginnen mit dem Import der benötigten Module. Die Hilfsklasse `TagStripper` erlaubt es uns, HTML-Tags aus den HTTP-Responses (die wir später weiterverarbeiten) zu entfernen. Deren `handle_data`-Funktion speichert den Seitentext ❶ in einer Member-Variablen. Wir definieren auch `handle_comment`, da wir auch die Wörter in den Entwicklerkommentaren in unserer Passwortliste festhalten wollen. Intern ruft `handle_comment` einfach nur `handle_data` ❷ auf (für den Fall, dass wir die Verarbeitung des Seitentextes ändern wollen).

Die `strip`-Funktion übergibt den HTML-Code an die Basisklasse `HTMLParser` und liefert die resultierende Textseite zurück ❸, die sich später noch als nützlich erweisen wird. Der Rest entspricht genau dem Anfang des *bhp_bing.py*-Skripts, das wir gerade abgeschlossen haben. Erneut wollen wir die Burp-UI um ein Kontextmenü-Element erweitern. Das einzig Neue ist hier, dass wir unsere Wortliste in einem `set` (also einer »Menge«) speichern, wodurch sichergestellt wird, dass unsere Wortliste keine Duplikate enthält. Wir initialisieren unser `set` mit jedermanns Lieblingspasswort »password« ❹, nur um sicherzugehen, dass es tatsächlich in unserer Liste landet.

Nun fügen wir die Logik ein, die den gewählten HTTP-Traffic von Burp übernimmt und in unsere Basis-Wortliste umwandelt.

```
    def wordlist_menu(self,event):

        # Wahl des Benutzers abrufen
        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)

        # Nicht-Text-Responses überspringen
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)
```

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

```
            for word in words:

                # Lange Strings herausfiltern
                if len(word)  <= 12:
❻                   self.wordlist.add(word.lower())

            return
```

Als Erstes definieren wir die Funktion `wordlist_menu`, die unseren Menüklick-Handler darstellt. Sie speichert für später den Namen des Hosts ❶, ruft dann die HTTP-Response ab und übergibt diese an unsere `get_words`-Funktion ❷. Von dort filtert `get_words` die Header aus dem Body und stellt sicher, dass wir nur textbasierte Antworten verarbeiten ❸. Unsere `TagStripper`-Klasse ❹ entfernt den HTML-Code aus dem Rest des Seitentextes. Wir nutzen einen regulären Ausdruck, um alle Wörter zu finden, die mit einem alphabetischen Zeichen beginnen, auf das zwei oder mehr »Wort«-Zeichen folgen ❺. Nachdem wir zu lange Wörter ausgeschlossen haben, sichern wir die Wörter in Kleinbuchstaben in `wordlist` ❻.

Runden wir nun unser Skript mit der Fähigkeit ab, die erzeugte Wortliste zu verändern und auszugeben.

```
        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
```

Sehr nett! Die `mangle`-Funktion nimmt ein Grundwort und verwandelt es in eine Reihe möglicher Passwortkandidaten, indem es einige gängige »Strategien« zur Passwortgenerierung nutzt. In diesem einfachen Beispiel erzeugen wir eine Liste von Endungen, die an das Basiswort angehängt werden, darunter auch das aktuelle Jahr ❶. In einer Schleife gehen wir dann jede Endung durch und hängen Sie an das Grundwort an ❷, um die eindeutigen Passwortkandidaten zu erzeugen. Zusätzlich geben wir eine großgeschriebene Variante des Grundwortes in einer weiteren Schleife aus. In der Funktion `display_wordlist` geben wir einen »John

the Ripper«-artigen Kommentar aus ❸, der uns daran erinnert, welche Websites zur Generierung der Wortliste verwendet wurden. Dann erzeugen wir aus der Wortliste unsere Passwortkandidaten und geben die Ergebnisse aus. Höchste Zeit, unser Baby laufen zu lassen.

Die Probe aufs Exempel

Klicken Sie in Burp den **Extender**-Tab und anschließend den **Add**-Button an und verwenden Sie dann die bereits bei den anderen Erweiterungen genutzte Prozedur, um die Wortlisten-Erweiterung zu integrieren. Sobald sie geladen ist, gehen Sie mit dem Browser auf *http://testphp.vulnweb.com/*.

Wählen Sie die Website im Site-Map-Bereich mit der rechten Maustaste aus und klicken Sie dann auf **Spider this host**, wie in Abbildung 7–12 zu sehen.

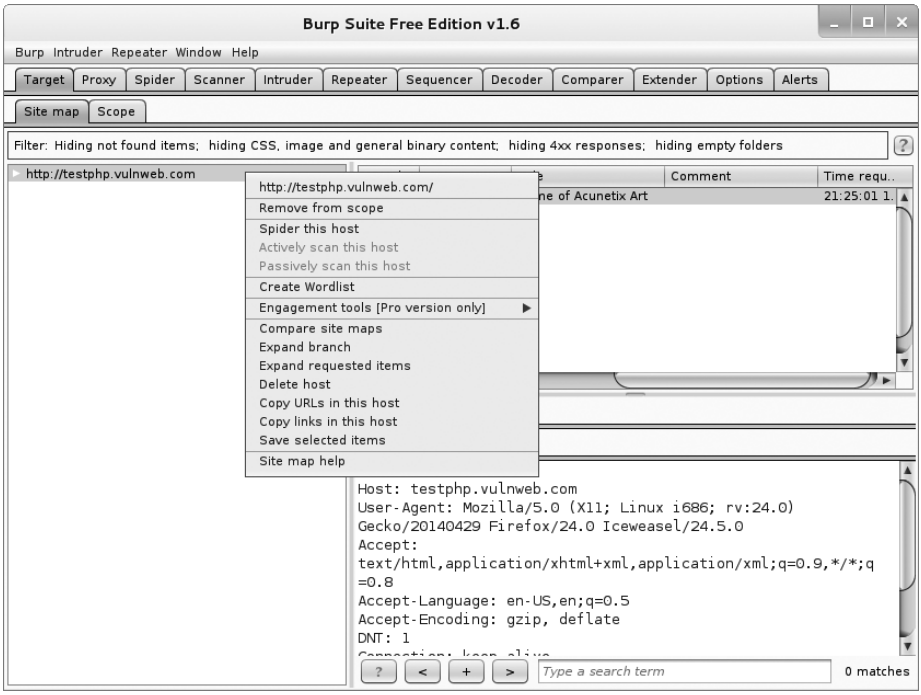

Abb. 7–12 *Host-Spidering mit Burp*

Nachdem Burp alle Links der Zielsite besucht hat, wählen Sie alle Requests im oberen rechten Bereich aus, öffnen durch einen Rechtsklick das Kontextmenü und wählen **Create Wordlist** (siehe Abbildung 7–13).

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4

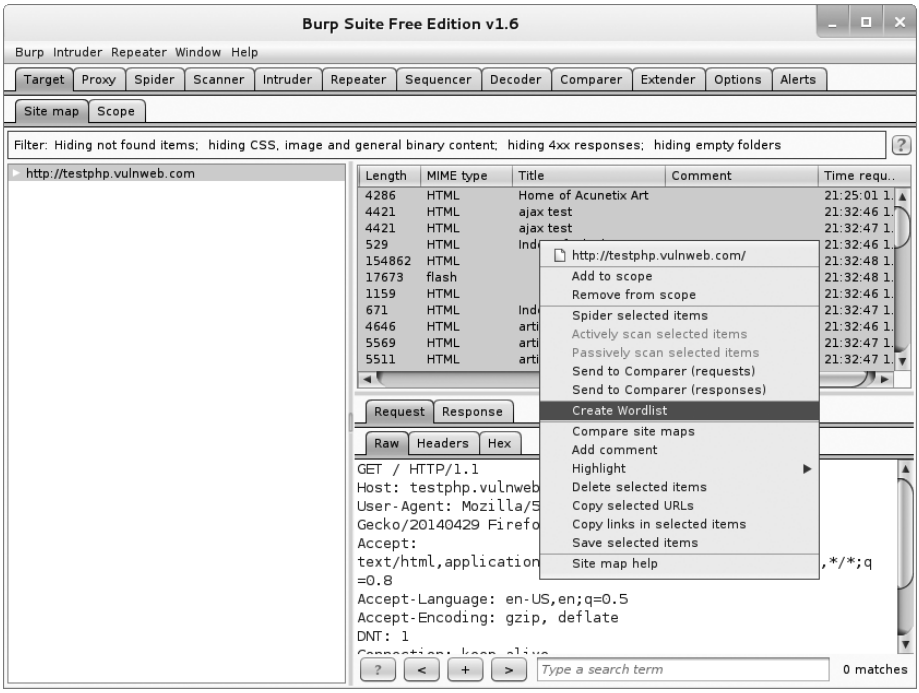

Abb. 7–13 *Requests an die BHP-Wortlisten-Erweiterung senden*

Sehen Sie sich nun den Output-Tab in der Erweiterung an. In der Praxis würden wir diese Ausgabe in einer Datei speichern, doch zu Demonstrationszwecken geben wir die Wortliste, wie in Abbildung 7–14 zu sehen, in Burp aus.

Sie können nun den Burp Intruder mit dieser Liste füttern, um den eigentlichen Passwortangriff zu starten.

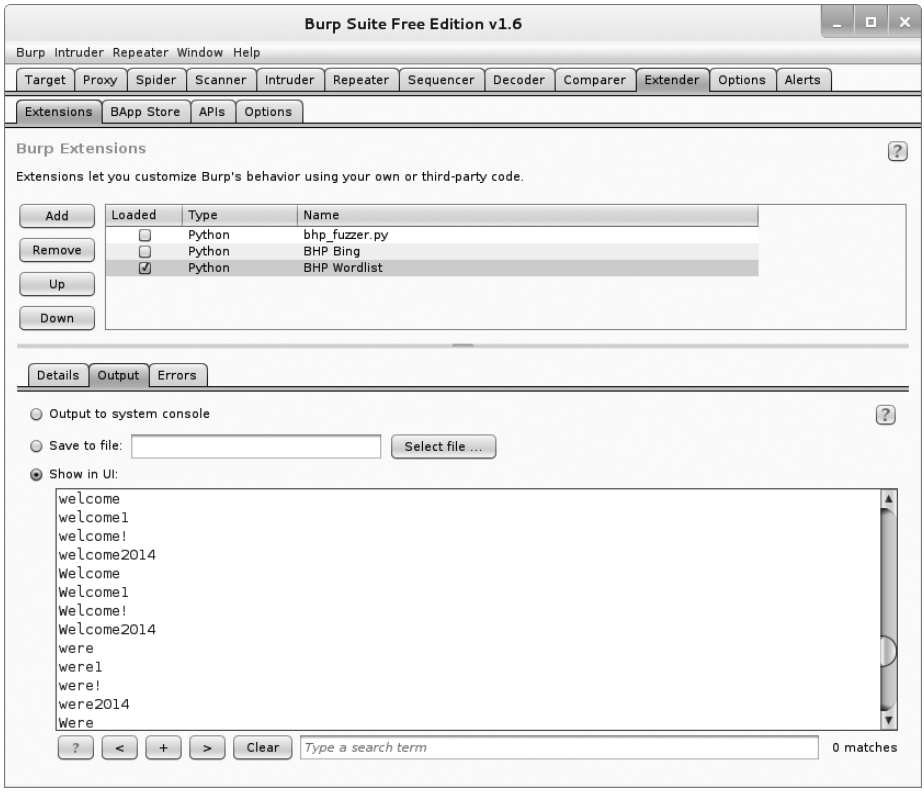

Abb. 7–14 *Passwortliste basierend auf dem Inhalt der Zielwebsite*

Wir haben nun einen kleinen Teil der Burp-API kennengelernt. Wir sind in der Lage, eigene Nutzdaten für unsere Angriffe zu erzeugen, und können Erweiterungen entwickeln, die mit der Burp-Benutzerschnittstelle interagieren. Während eines Penetrationstests werden Sie häufig auf spezifische Probleme oder Automatisierungsanforderungen stoßen. Die Burp-Extender-API bietet da eine ausgezeichnete Möglichkeit, sich aus dieser Ecke herauszumanövrieren. Zumindest erspart es Ihnen ein ständiges Kopieren und Einfügen der abgefangenen Daten in ein anderes Tool.

In diesem Kapitel haben wir Ihnen gezeigt, wie Sie ein ausgezeichnetes Erkundungstool in Ihre Burp-Werkzeugkiste integrieren können. Im Moment ruft die Erweiterung nur die ersten 20 Ergebnisse von Bing ab. Ihre Übungsaufgabe besteht also darin, zusätzliche Requests einzufügen, um auch wirklich alle Ergebnisse abzurufen. Dazu müssen Sie sich ein wenig in die Bing-API einlesen und Code entwickeln, der die größeren Ergebnismengen verarbeiten kann. Natürlich könnten Sie den Burp Spider dann anweisen, jede neu entdeckte Website abzugrasen und automatisch nach Sicherheitslücken zu suchen!

Justin Seitz, Mehr Hacking mit Python, dpunkt.verlag, ISBN 978-3-86490-286-4