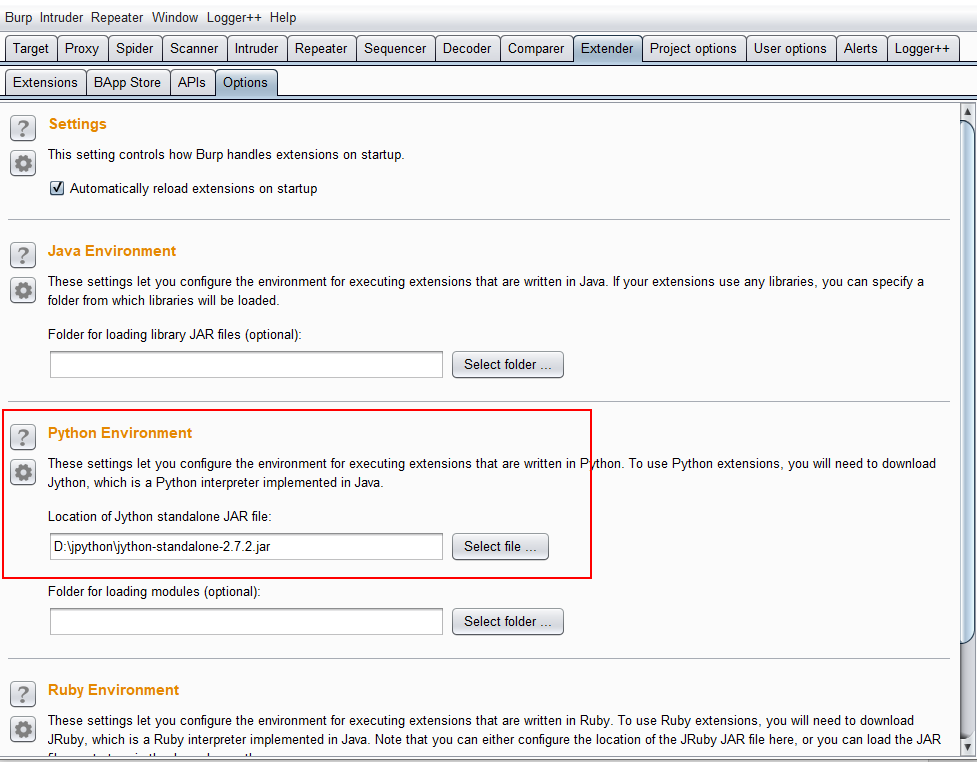
## Webfuzz插件

在实施Web应用测试的时候，你可以使用Burp中的很多工具。一般来说，你可以通过代理截取全部的请求流量，当你看到一个有意思的请求时，你可以将这个请求包传递给另一个Burp工具。我经常利用的一个简单技巧是将请求包发送给Repeater工具，这个工具可以让我重放Web流量，当然，要在重放之前人为修改感兴趣的地方。为了在查询页面参数的过程中更自动化地进行攻击，你可以将请求包发送给Intruder工具，这个工具会自动判断Web流量中哪些地方需要修改，之后允许你使用各种攻击引起错误提示或者梳理出漏洞。Burp扩展工具可以与BurpSuite中的工具通过多种方式进行交互，在下面的例子中，我们将直接在Intruder工具里添加新的功能。

### 安装jython

获取Jython（一个用Java编写的Python解释器)JAR独立文件；你可以在No Starch官方网站找到这个JAR文件及本书相关的源代码（http://wwwnostarch.com/blackhatpython/)或者直接访问Jython的官方网站htutp:/wwwjython.org/downloads.html,选择Jython2.7独立安装文件。别让文件名欺骗你，这就是一个JAR文件。将这个JAR文件保存在你熟悉的位置，比如桌面。

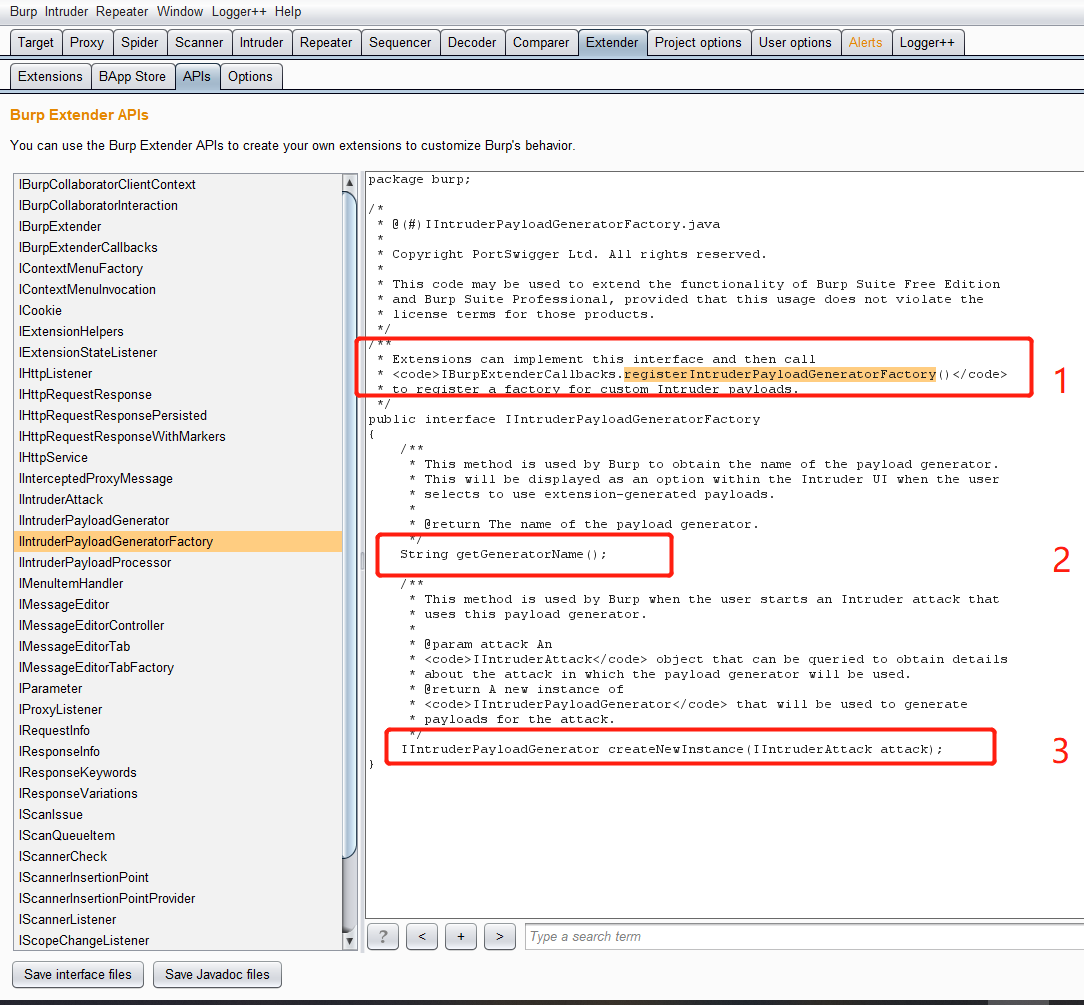
现在将Burp指向我们的Jython解释器。单击Extender标签，然后单击Options 按钮。在Python环境部分，选择JythonJAR文件的路径，如图所示。



### 插件实现

我的第一直觉是查看BurpAPI文档，判定哪些Burp类需要在编写工具时进行扩展。你可以通过单击Extender标签，然后单击后面的APIs标签查看相关文档。由于都是Java的风格，因此有点让人畏惧。首先，我们会注意到开发Burp的人员对每一个类都进行了适当的命名，这样我们就可以轻松地找到从哪里开始。

在这个例子中，我们需要在利用Intruder开展攻击时对Web请求进行模糊测试，我找到了IIntruderPayloadGeneratorFactory和IIntruderPayloadGenerator类。让我们查看下文档对IIntruderPayloadGeneratorFactory类的介绍：



文档的第一部分①告诉我们扩展工具需要在Burp上正确注册。我们将扩展Burp的主类和IIntruderPayloadGeneratorFactory类。接下来，Burp希望在我们的主类中使用两个函数。由Burp调用的getGeneratorName函数②获取我们的扩展工具的名字，我们希望它调用成功返回一个字符串。

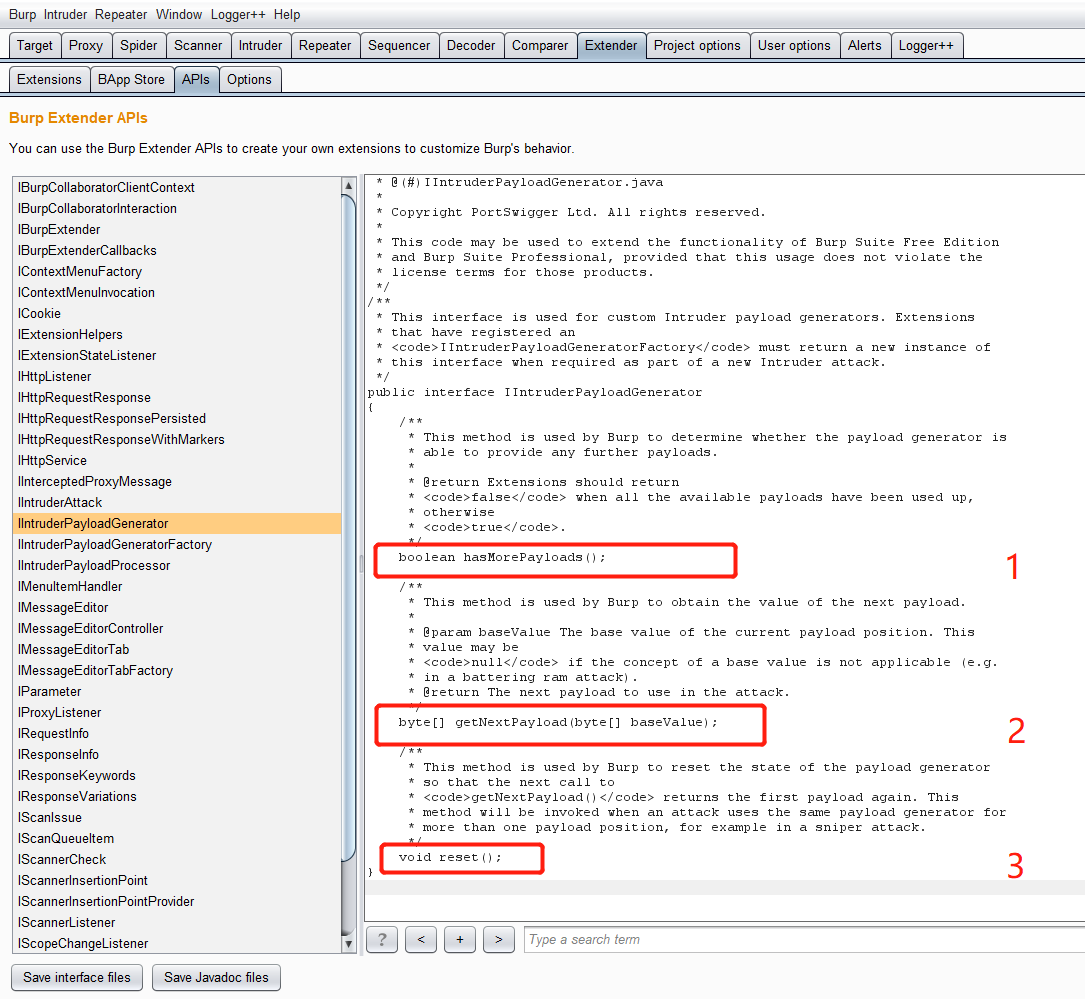
createNewInstance 函数③返回一个IIntruderpayloadGenerator类型的对象，IIntruderPayloadGenerator是需要我们创建的第二个类。

现在我们开始编写实际的Python代码以满足上面的要求，之后我们查看如何将IIntruderPayloadGenerator类添加进来。打开一个新的Python文件，命名为web\_fuzze:py并输入如下代码：

# 1. 导入IBurpExtender类,这是每一个Burp扩展工具时必须使用的类  
from burp import IBurpExtender  
from burp import IIntruderPayloadGeneratorFactory  
from burp import IIntruderPayloadGenerator  
from java.util import List, ArrayList  
import random  
  
  
# 2. 定义自己的BurpExtender类  
class BurpExtender(IBurpExtender, IIntruderPayloadGeneratorFactory):  
 def registerExtenderCallbacks(self, callbacks):  
 self.\_callbacks = callbacks  
 self.\_helpers = callbacks.getHelpers()  
 callbacks.setExtensionName("web\_fuzz")  
 # 3. 使用registerIntruderPayloadGeneratorFactory 函数注册BurpExtender类，  
 # 这样Intruder工具才能生成攻击载荷。  
 callbacks.registerIntruderPayloadGeneratorFactory(self)  
 return  
  
 # 4. getGeneratorName函数,让它返回载荷生成器的名称。  
 def getGeneratorName(self):  
 return "web\_fuzz"  
 # 5. createNewInstance函数接收攻击相关的参数，  
 # 返回IIntruderPayloadGenerator类型的实例，我们将它命名为BHPFuzzer。  
 def createNewInstance(self, attack):  
 return BHPFuzzer(self, attack)

这是扩展工具所需要的一个简单架构，它满足了前面我们提到的Burp对扩展工具的一些基本要求。首先，我们导入了IBurpExtender类①，它是编写每一个Burp扩展工具时必须使用的类，紧接着我们为创建Intruder载荷生成器导入必要的类。第二步，我们定义自己的BurpExtender类②，它继承和扩展了IBurpExtender 类和IIntruderPayloadGeneratorFactory类。之后我们使用registerIntruderPayloadGeneratorFactory函数③注册BurpExtender类，这样Intruder工具才能生成攻击载荷。第三步，我们部署getGeneratorName函数④让它返回载荷生成器的名称。最后一步，createNewInstance函数⑤接收攻击相关的参数，返回IIntruderPayloadGenerator类型的实例，我们将它命名为BHPFuzzer。

让我们查看一下IIntruderpayloadGenerator类的文档，了解它实现了什么样的功能。



好了！我们需要部署的基类包含三个函数。第一个函数是hasMorePayloads（），用来判定是否继续把修改后的请求发送回Burp Intruder。我们使用一个计数器来解决这个问题，一旦计数器达到我们设置的上限，程序返回False，这样不会继续生成模糊测试的请求。getNextPayload函数②接受你劫持的HTTP请求包中的原始负载作为参数，如果你在HTTP请求包中选择了多个攻击载荷参数，你将只能从模糊测试工具接收到字节数据（后面还要讨论这个问题）。这个函数允许我们模糊测试原始的数据后再返还给它本身，这样Burp就可以发送新的模糊测试参数。最后一个函数reset③，如果我们生成已知的模糊测试请求（一般是五个），匹配我们设计的Intruder标签中攻击载荷的位置，那么我们将逐次使用这五个模糊测试值。

我们的模糊测试工具不是非常挑剔，它会持续对每一个HTTP请求进行随机化的模糊测试。现在让我们看看如何在Python中实现它。在bhp fiuczerpy底部添加以下代码：

# 6. 扩展继承IIntruderPayloadGenerator类  
class BHPFuzzer(IIntruderPayloadGenerator):  
 def \_\_init\_\_(self, extender, attack):  
 self.\_extender = extender  
 self.\_helpers = extender.\_helpers  
 self.\_attack = attack  
 print "web\_fuzz"  
 # 7. 设置计数器  
 self.max\_payloads = 1000  
 self.num\_payloads = 0  
   
 return  
   
 # 8. 检查模糊测试时迭代的数量是否到达上限  
 def hasMorePayloads(self):  
 print "hasMorePayloads called."  
 if self.num\_payloads == self.max\_payloads:  
 print "No more payloads."  
 return False  
 else:  
 print "More payloads. Continuing."  
 return True  
   
 # 9. 负责接收原始的HTTP载荷，这里就是进行模糊测试的地方。  
 def getNextPayload(self,current\_payload):  
 # 10. current\_payload变量是数组格式，我们需要将它转换成字符串  
 payload = "".join(chr(x) for x in current\_payload)  
 # 11. 传递给模糊测试的函数mutate\_payload  
 payload = self.mutate\_payload(payload)  
 # 12. 将num\_payloads变量的值增加  
 self.num\_payloads += 1  
 return payload  
   
 def reset(self):  
 self.num\_payloads = 0  
 return

我们从定义自己的BHPFuzzer类⑥开始，它扩展了IIntruderPayloadGenerator类。我们定义了需要的类变量，还添加了max\_payloads⑦和num\_payloads变量，它们用来对模糊测试的过程进行追踪，让Burp了解模糊测试完成的时间。

当然，你也可以让扩展工具一直运行下去，但是对于测试来说，我们在此处进行了限制。接下来，我们部署hasMorePayloads函数⑧，该函数检查模糊测试时迭代的数量是否到达上限。你可以通过控制返回True让扩展工具一直运行下去。getNextPayload函数⑨负责接收原始的HTTP载荷，这里就是进行模糊测试的地方。current\_payload变量是数组格式，我们需要将它转换成字符串⑩，然后传递给模糊测试的函数mutate\_payload⑪。之后，我们将num\_payloads变量⑫的值增加，然后返回修改之后的载荷。最后一个函数是reset，它没有做任何工作。

现在，我们深入这个全世界最简单的模糊测试函数，修改它的核心内容。

由于这个函数只关心当前的HTTP负载，如果请求包中含有一些特殊协议：如载荷开始时包含CRC校验或者字符长度，那么你就可以在函数返回前在函数内部计算和添加这些值，这样的实现方式非常灵活。将如下代码添加到bhpfuezerpy中，确保mutate\_payload函数已经加入BHPFuzzer类中：

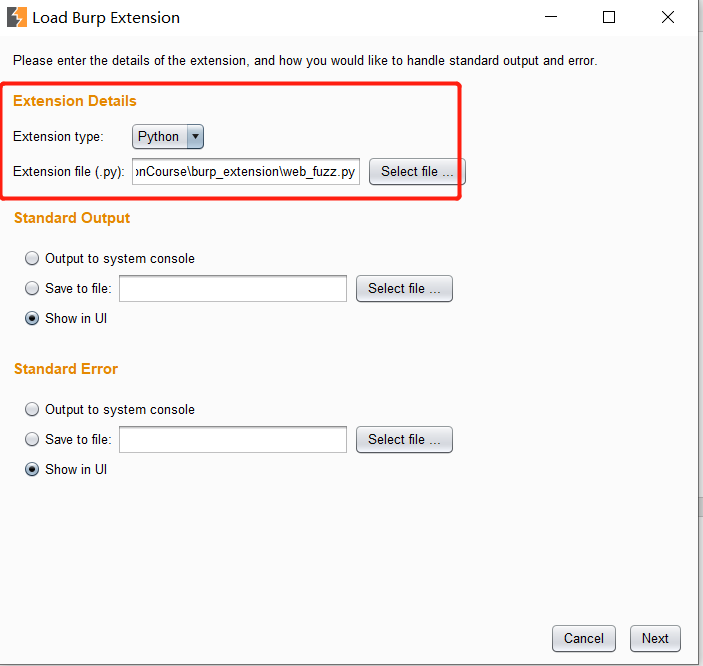
# 模糊测试函数  
def mutate\_payload(self,original\_payload):  
 picker = random.randint(1,3)  
 offset = random.randint(0,len(original\_payload)-1)  
 payload = original\_payload[:offset]  
  
 if picker == 1:  
 payload += "'"  
  
 if picker == 2:  
 payload += "<script>alert('xss');</script>";  
  
 if 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]  
  
 payload += original\_payload[offset:]  
 return payload

这个模糊测试工具的结构非常清晰。我们从三个变形器中随机选取：用一个单引号进行SQL注入测试，一个跨站测试，之后从变形器随机选取一个原始的攻击载荷并随机重复。

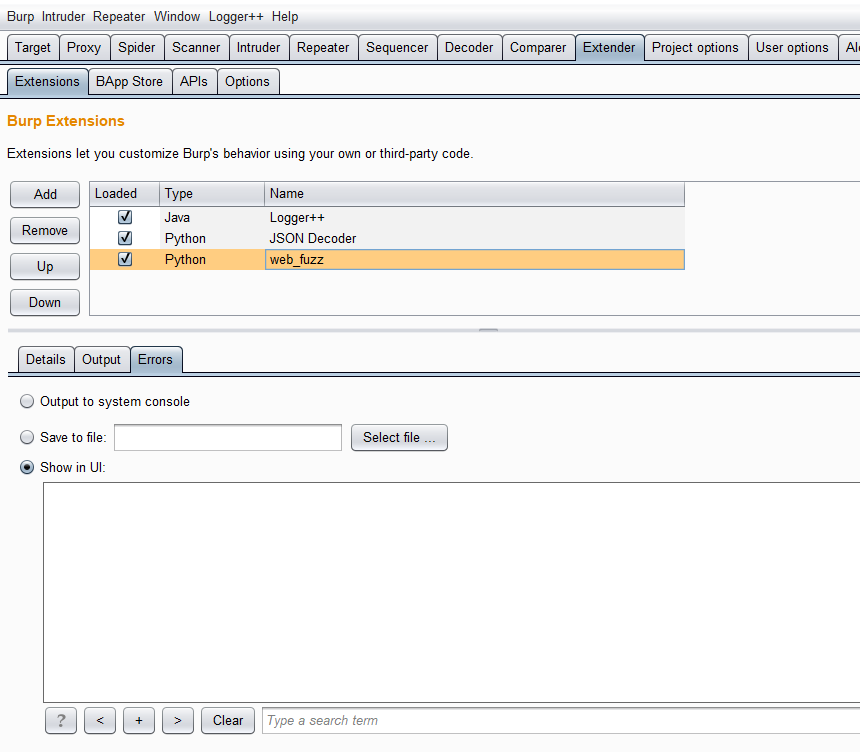
### Burp加载插件

现在，我们有一个Burp Intruder扩展工具可以使用了，让我们查看它如何加载。

首先，我们需要将扩展工具添加到Burp中并查看是否有错误提示。在Burp中单击Extender标签，然后单击Add按钮。界面显示你可以在Burp中指定模糊测试器。请确保你的设置与截图所示一致。



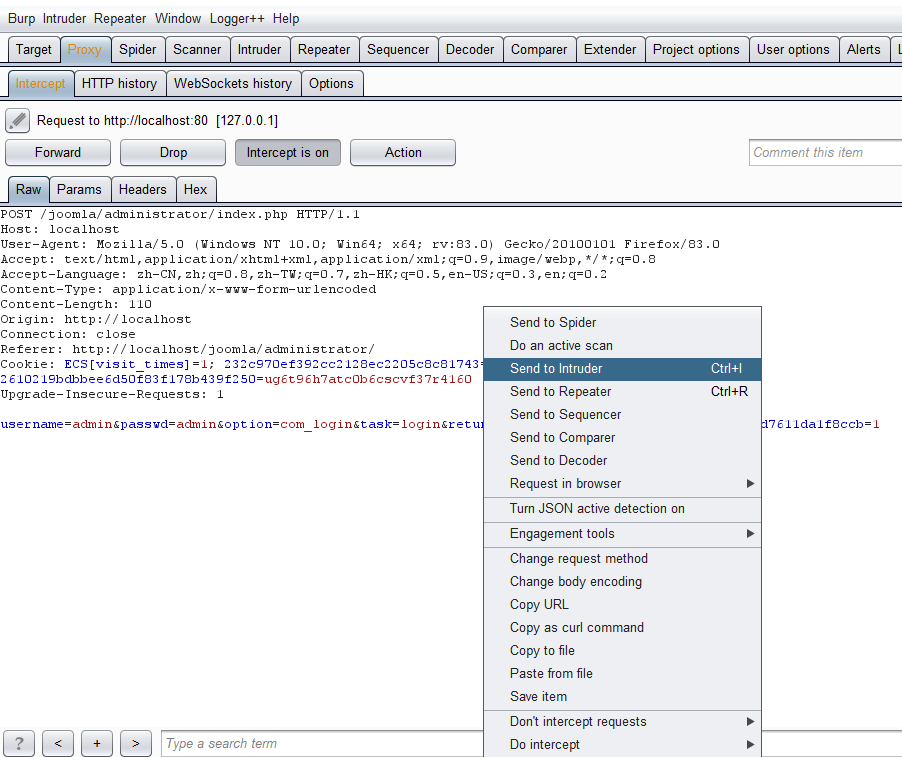
单击Next按钮，这样Burp就开始加载我们的扩展工具了。如果一切顺利，那么Burp会提示扩展工具加载成功；如果存在错误，则单击Errors 标签，调试错误，然后单击Close按钮。现在你的扩展界面应该与图所示类似。如果报错了，删除掉代码中的全部汉字和注释。



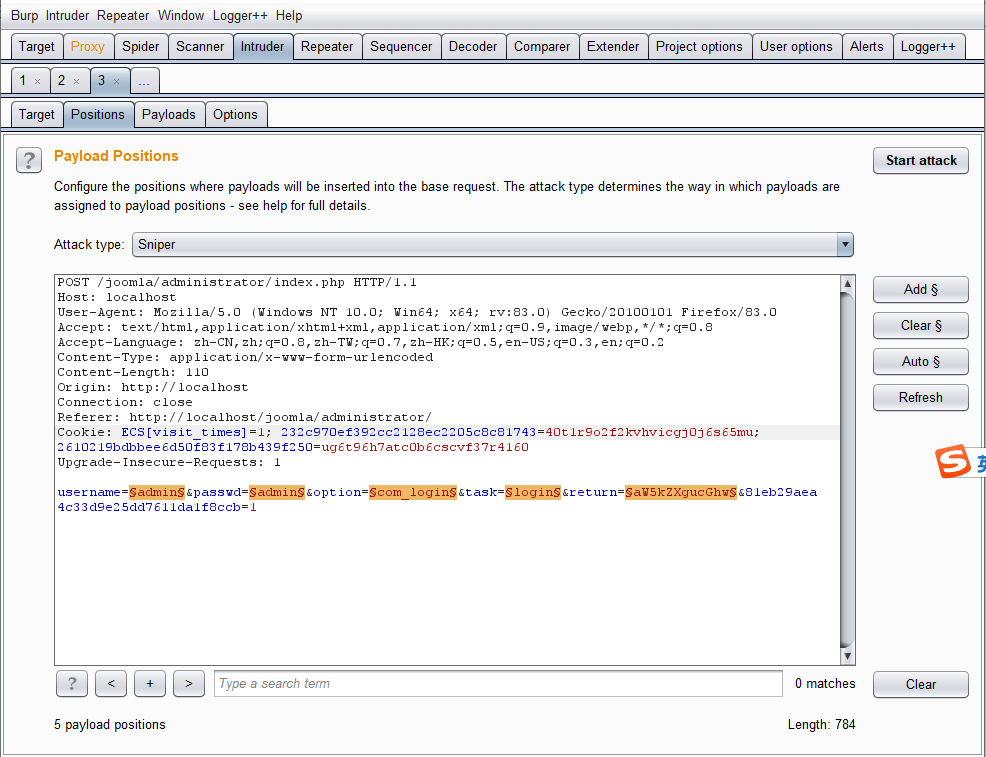
可以看到，我们的扩展工具已经加载，Burp识别出已经注册了一个Intruder载荷生成器。现在我们可以使用扩展工具在现实中进行攻击。确保我们已经将浏览器设置成从本地的8080端口转发以使用Burp代理。

### 运行测试

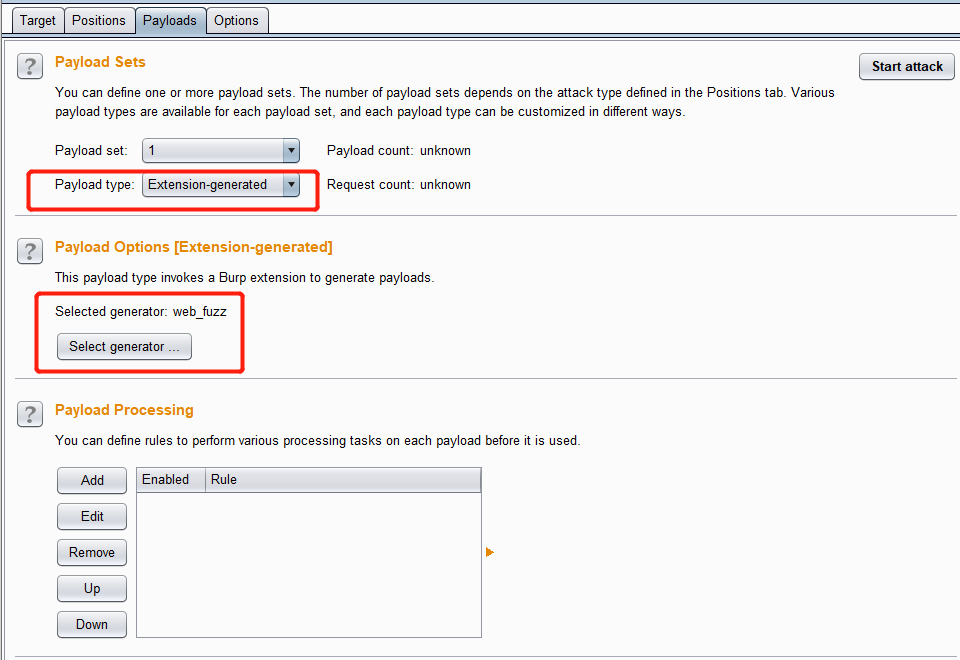
作为一个例子，我用burp抓到了joomla的登录包。截图展示了我是如何在Proxy标签的HTTP历史标签中查看这个请求的，鼠标右键单击可将这个请求发送给Intruder。



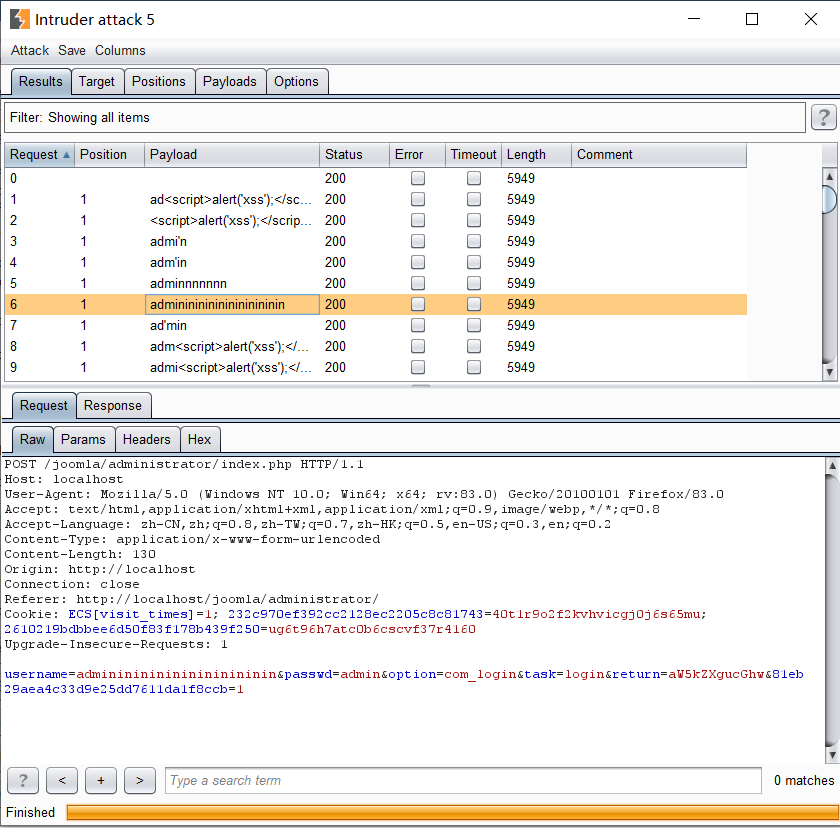
现在切换到Intruder标签并单击Positions标签，界面将高亮显示每一个请求的参数。这就是Burp识别出来我们需要进行模糊测试的地方。你可以选择不同的载荷范围进行尝试，当然也可以选择全部的载荷进行模糊测试，在这个例子中，我们让Burp决定需要进行模糊测试的攻击载荷。图清晰地显示了高亮后的载荷是什么样子。



在当前屏幕中单击Payloads标签，然后在Payload type下拉菜单中选择Extension-generated 选项。在载荷选项（Payload Options)部分，单击 Selectgenerator..按钮并从下拉菜单中选择web\_fuzz选项。



现在我们已经准备好发送请求了，在Burp顶部的菜单栏，单击Intruder之后选择Start Attack选项，之后就开始对每一个请求进行模糊测试，你可以快速查看结果。

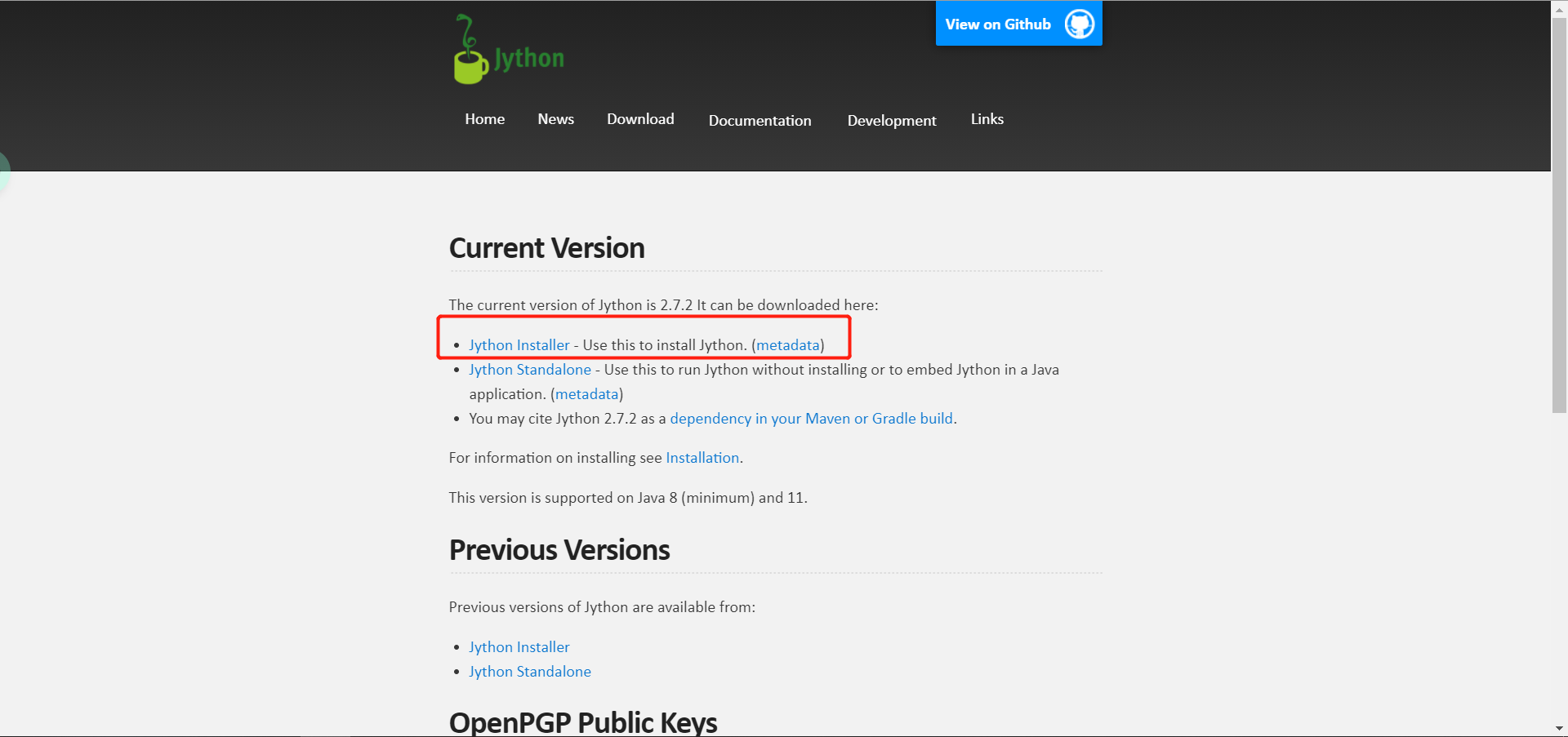


## Joomla爆破插件

## Burp插件-python调试方法

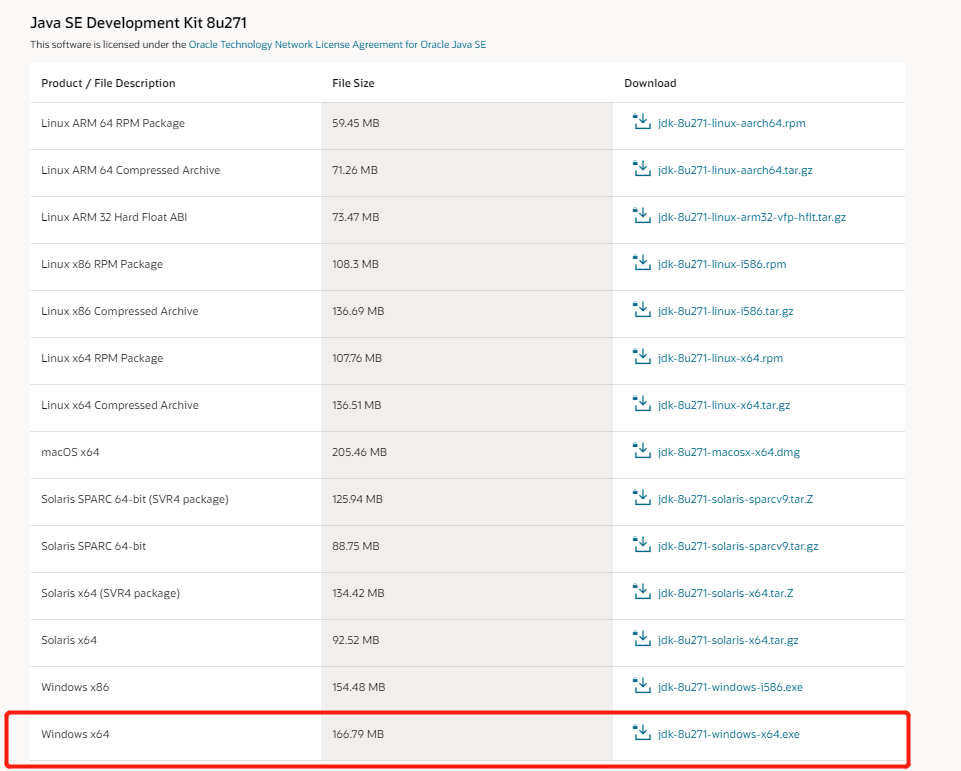
### 准备环境

1. 安装Idea专业版，是java编程语言开发的集成环境。可以通过配置jython来编写python代码。下载地址https://www.jetbrains.com/idea/download/#section=windows
2. Jython安装版，之前burp插件安装的是独立版。我们需要jython 的解释器所以需要安装jython的安装版。下载地址https://www.jython.org/download



1. 安装java。我的系统是windows64位的，所以我下载Windows x64的jdk安装版。

下载地址https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html



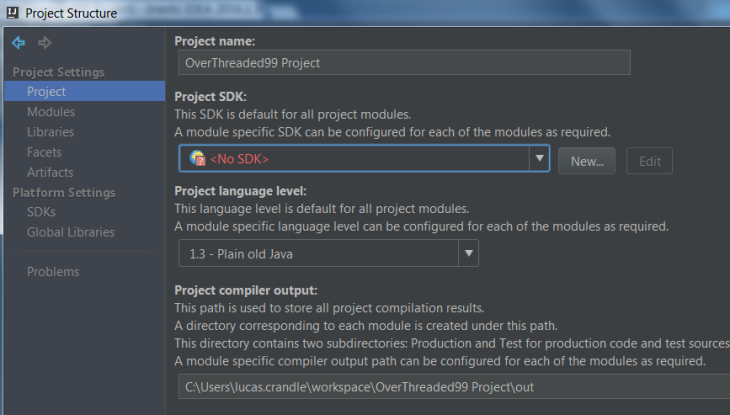
1. Burp的python版本的api。（这个不是必须的，有这个可以让编写代码时没有报错，要想运行burp插件，最后要删除掉这部分的。）

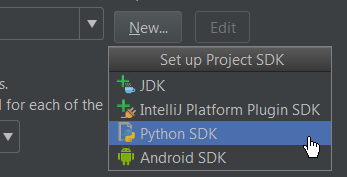


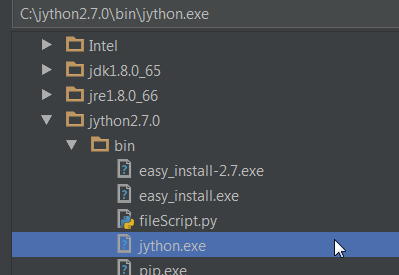
1. Burp和burp加载器。

### 配置环境

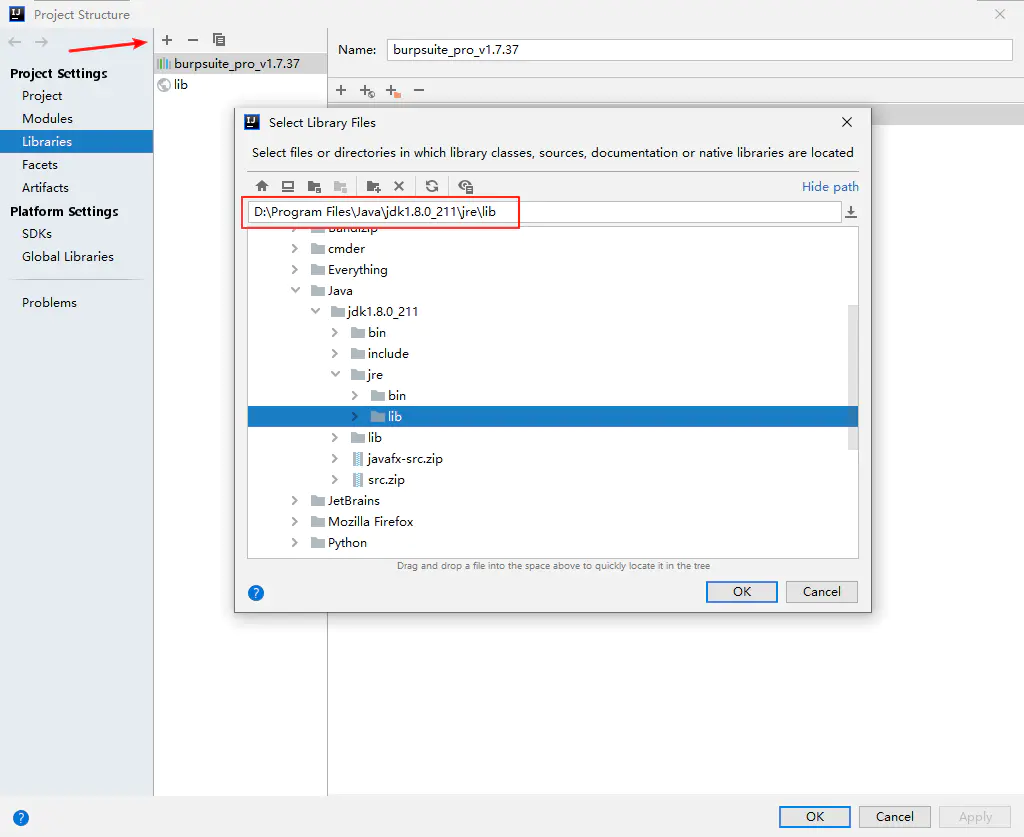
1. 打开idea，新建空工程
2. 安装Python插件：File->Settings->Plugins，搜索Python插件并安装
3. 设置SDK：File->Project Structure->project->选择Jython.exe（这个jython的安装版才会有）作为SDK





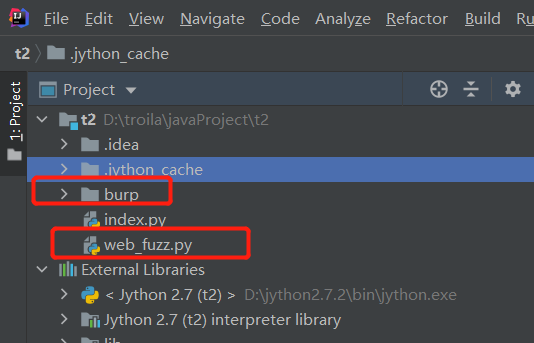


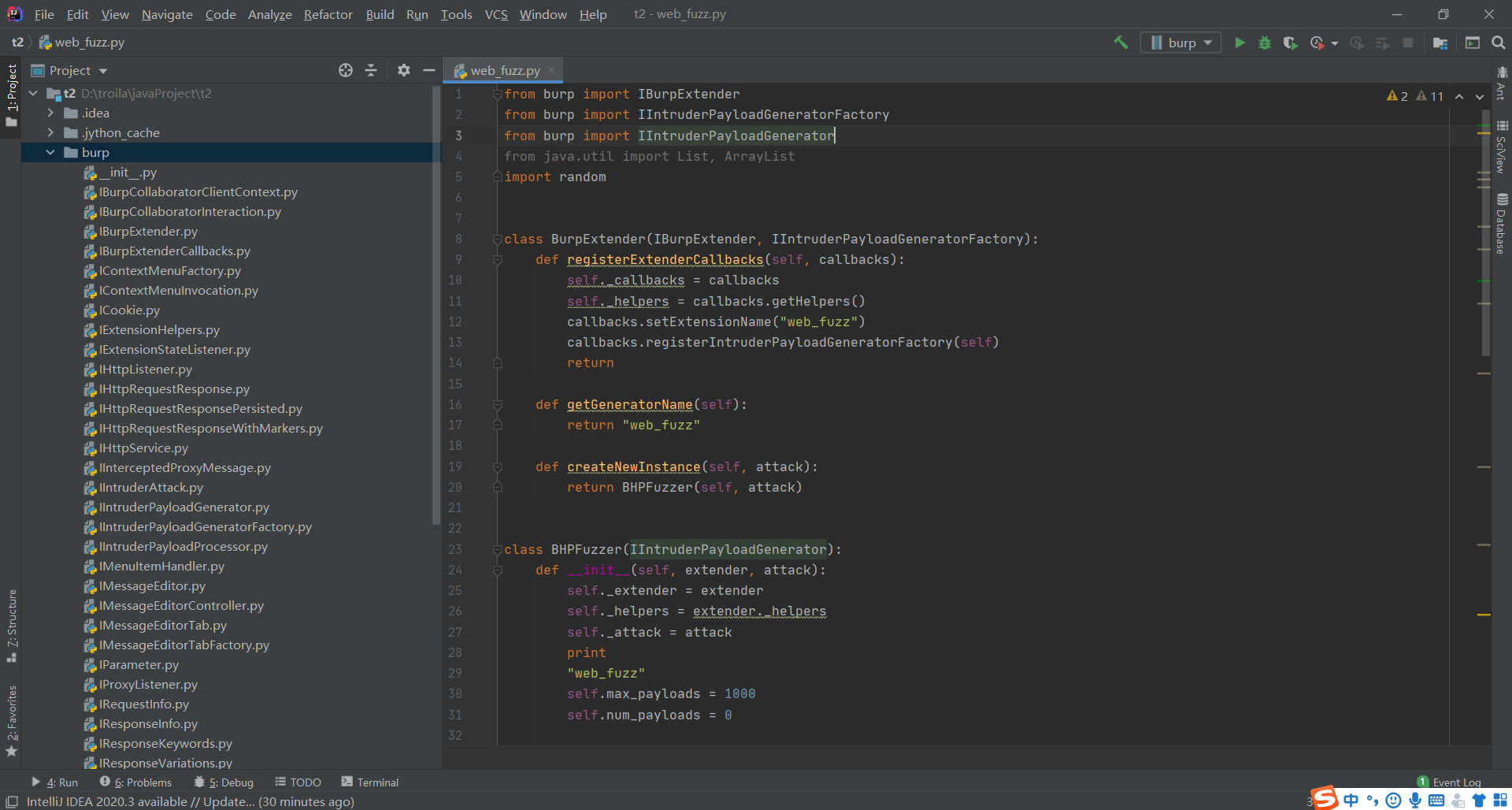
1. 添加Libraries：File->Projcet Structure->Libraries->点击+按钮（New Project Library）->选择Java->选择JDK里JRE里的lib目录。



1. 将burp的python版本的api加入到项目，复制之前编写的web\_fuzz.py插件到此项目。

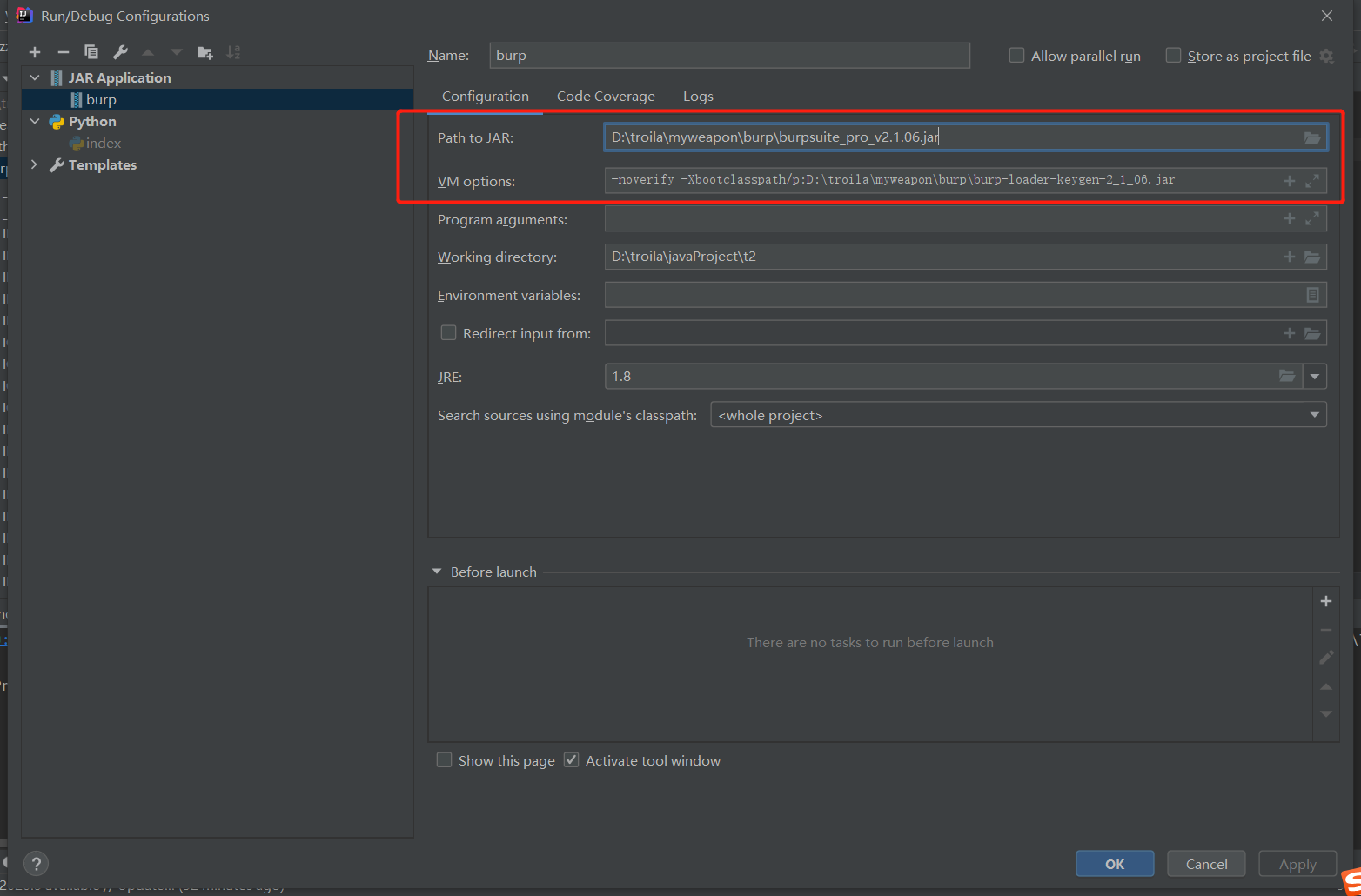
你会发现，此时我们的编写的python插件没有红色波浪线报错了。



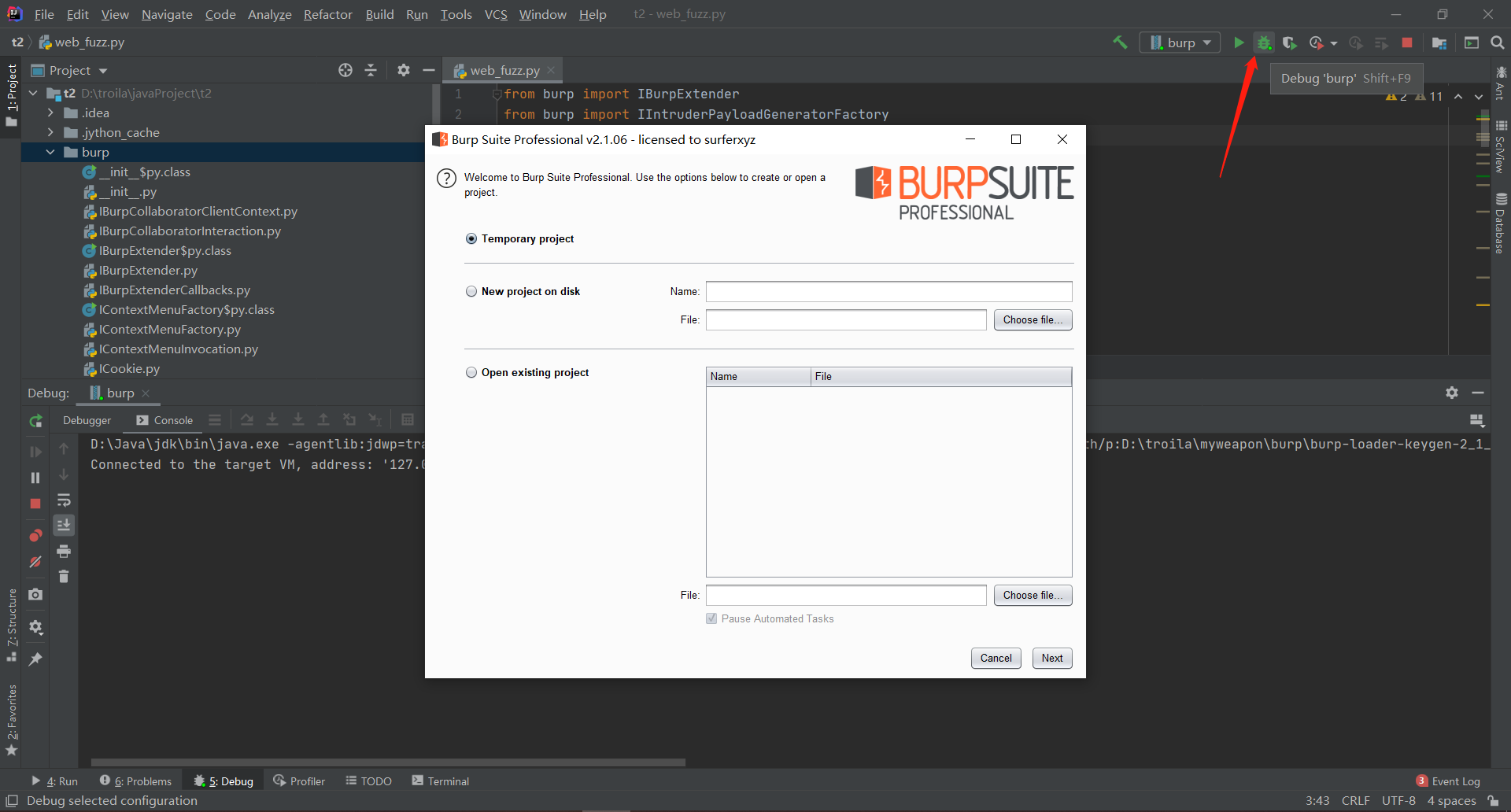


1. 配置burp。配置IDEA的Run/Debug Configurations，添加一项Jar Application类型的运行配置

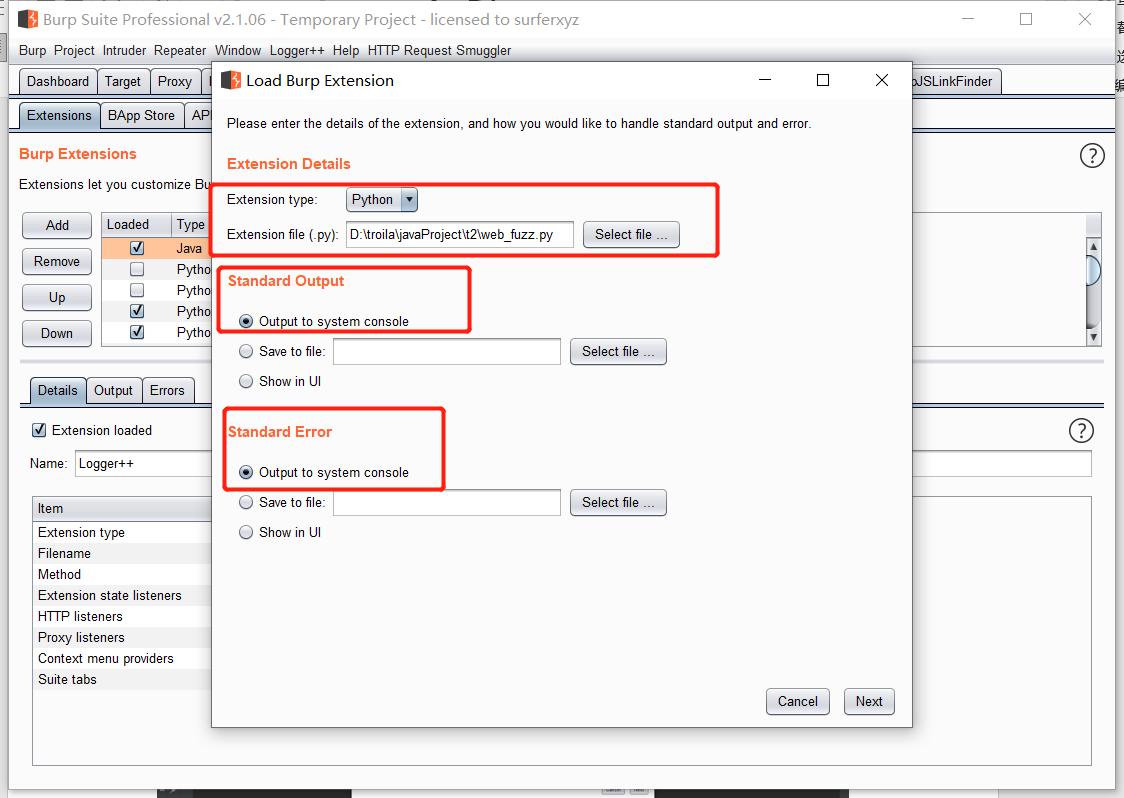
参数：-noverify -Xbootclasspath/p:D:\troila\myweapon\burp\burp-loader-keygen-2\_1\_06.jar



然后我们就可以通过idea来启动burp了。

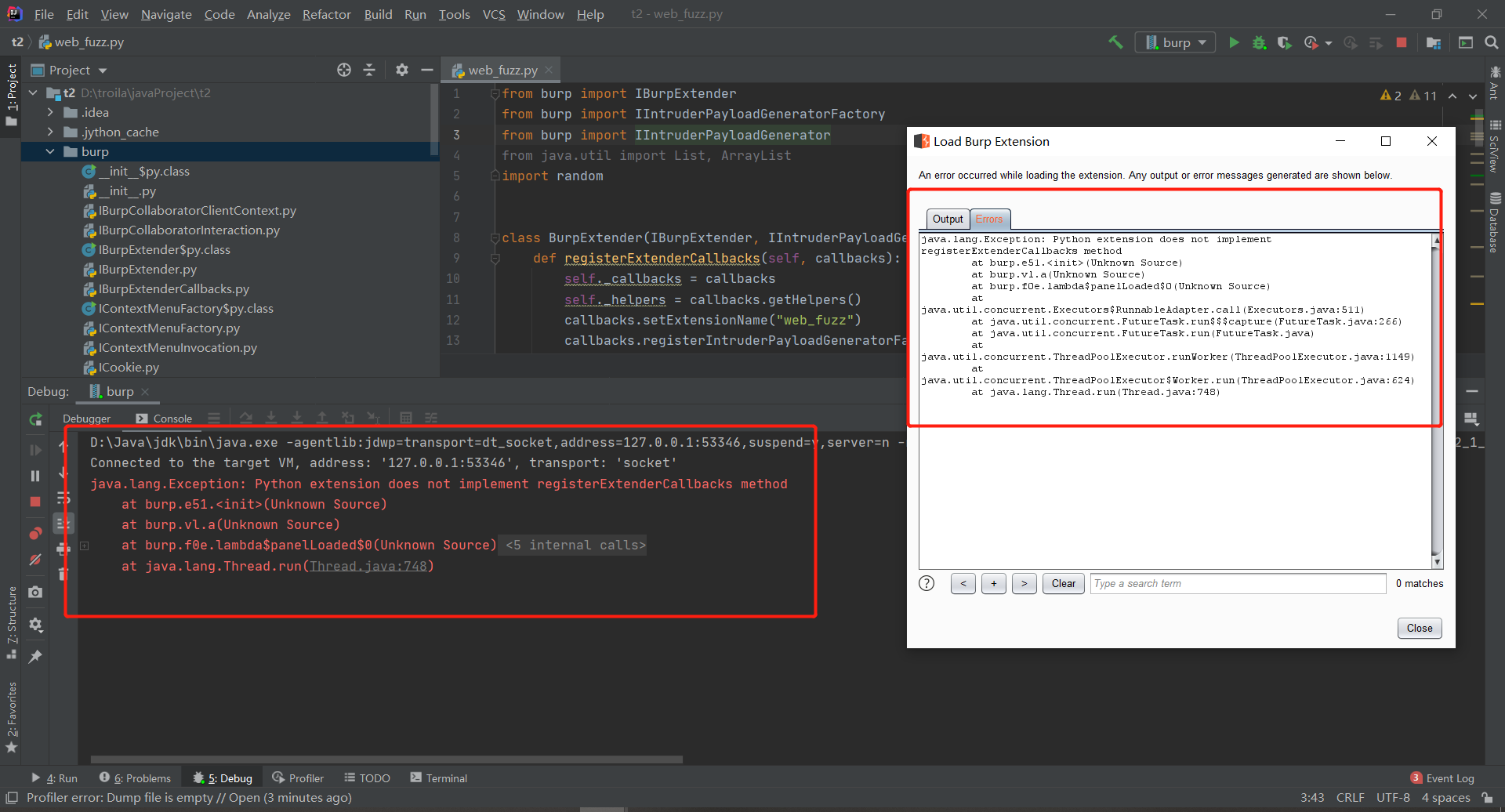


然后burpsuite添加自己的插件，并将插件的Output和Errors勾选为Output to system console

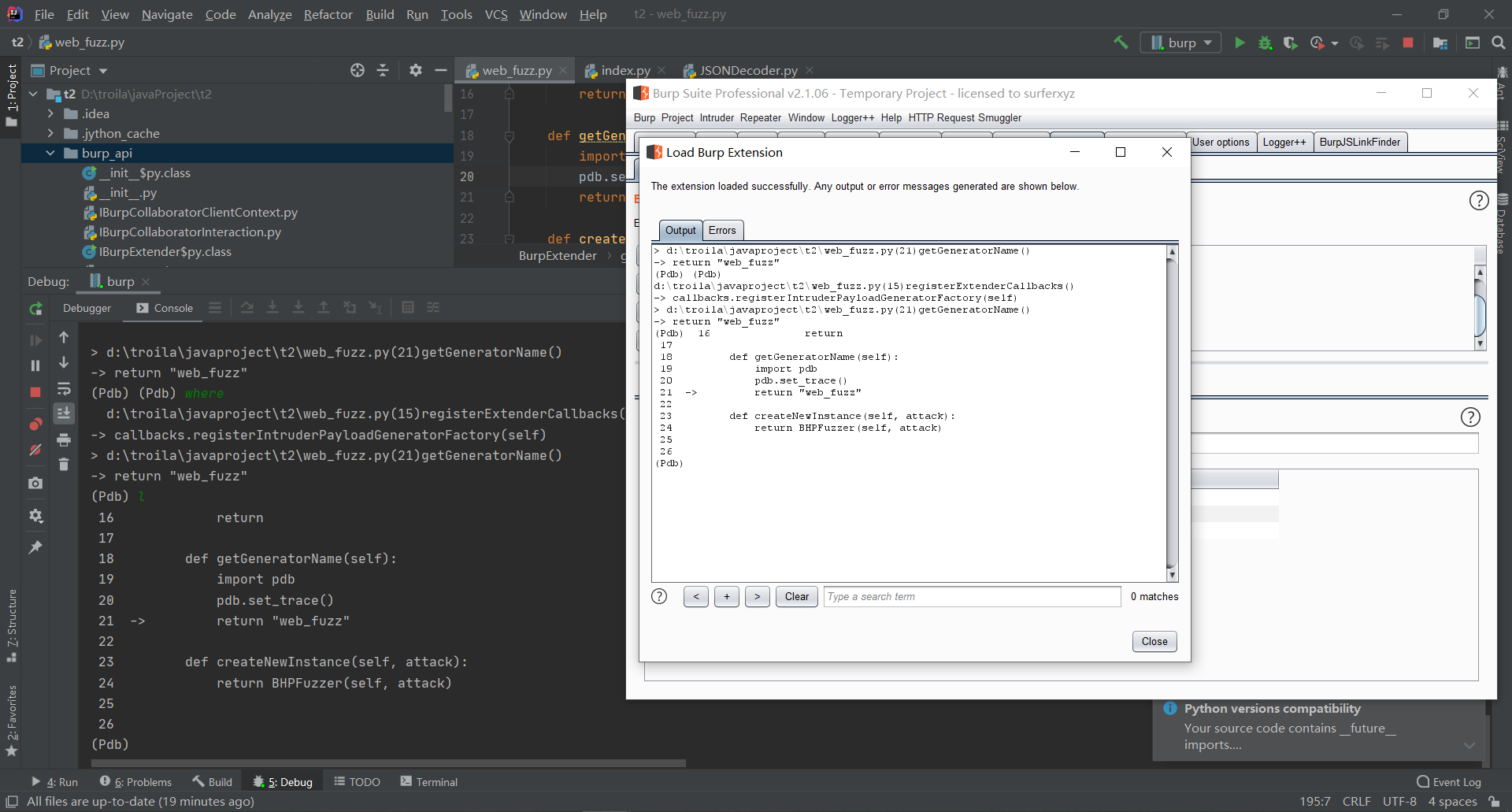


这样控制台就可以接收到输出信息了。

这时候会发现一个正常的插件代码在这里运行就会报错，那是因为burp包重名了。我们只需要删除掉burp文件夹，这时idea又开始有红色波浪线了。然后把py插件导入burp就发现正常了。所以我们编写python 代码时可以导入burp的py版本api，完成之后再删掉burp文件夹。



1. 设置断点。在需要调试的地方加import pdb; pdb.set\_trace()来下断点。当运行到pdb.set\_trace()时终端就出现pdb调试会话。



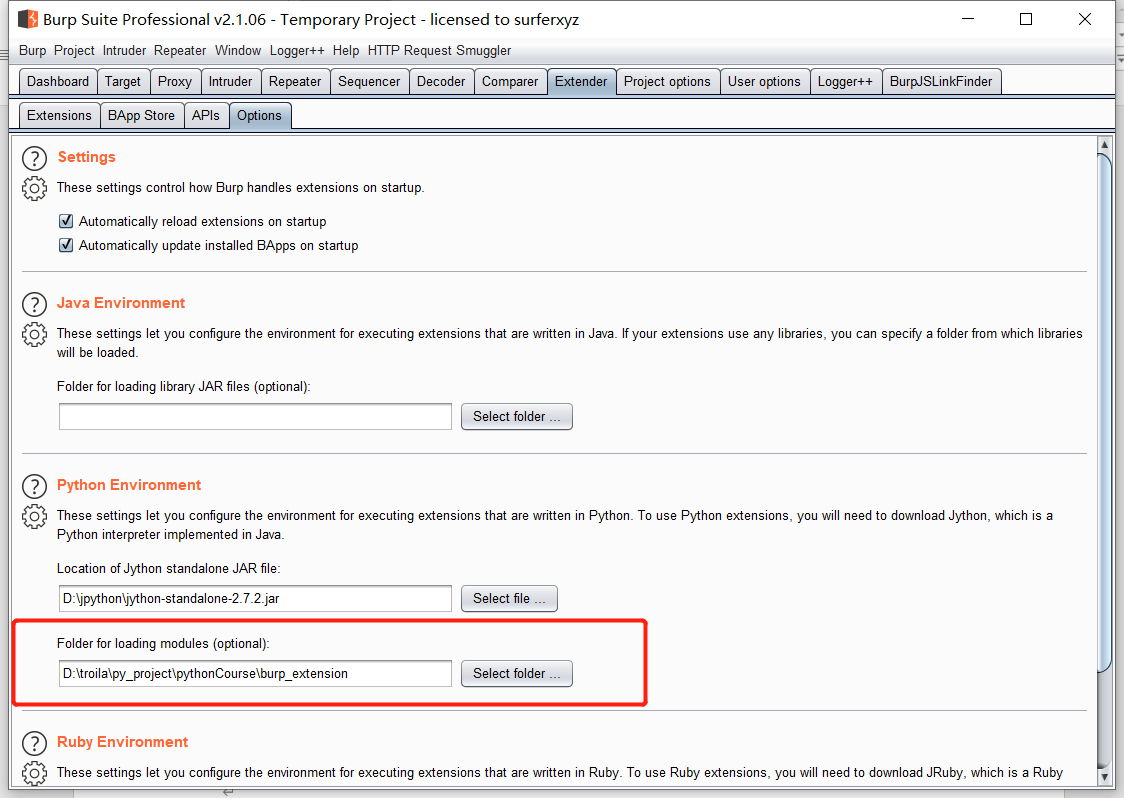
## Burp插件-框架逻辑分离

用Python编写和调试BurpSuite扩展，一件令人沮丧的事情是，即使我们做了最小的更改，也必须重新加载扩展程序。这个过程需要一些时间，并且因为我使用的是Jython，所以每次重新加载扩展时都会占用一些内存。

解决方法：，如果将新功能移动到第二个文件，则可以将其导入一次，然后根据需要频繁地重新加载它。

1. 指定模型加载目录

这个目录里放入我们要改动的文件，burp会自动从这个目录里查找要导入的模块，然后自动导入。



1. 编写框架必须的部分

这部分是burp插件所必须要实现的，然后在这个基础上导入UniqueParamValues模块，这个模块是我们的自定义功能。第三步就是实现这个模块。然后放在第一步提到的那个目录里。

Burp插件导入这个文件，逻辑实现在第三步的文件。

from burp import IBurpExtender  
from burp import IContextMenuFactory  
from burp import IExtensionHelpers  
from javax.swing import JMenuItem  
from java.awt.event import ActionListener  
from java.awt.event import ActionEvent  
from java.awt.event import KeyEvent  
import traceback  
# Burp is configured to look for python modules in c:\python27\lib.  
# If the following file exists in that directory, it will be loaded  
import UniqueParamValues  
  
  
class BurpExtender(IBurpExtender, IContextMenuFactory, ActionListener):  
 def \_\_init\_\_(self):  
 self.menuItem = JMenuItem('Print Unique Parameter Values')  
 self.menuItem.addActionListener(self)  
  
 def actionPerformed(self, actionEvent):  
  
 print "\*" \* 60  
 # Here is the reload. You can place this anywhere you wantm but you will  
 # most likely want to place this within an action (request recieved, menu  
 # item clicked, scanner started, etc).  
 reload(UniqueParamValues)  
 # This try statement, and the traceback included in the except, are what  
 # allowed me to finally get the trace information I needed to debug my  
 # issues. I highly recommned including these when developing Burp  
 # Extensions  
 try:  
 UniqueParamValues.getUniqueParams(self)  
 except:  
 tb = traceback.format\_exc()  
 print tb  
  
 # implement IBurpExtender  
  
  
 def registerExtenderCallbacks(self, callbacks):  
 # keep a reference to our callbacks object (Burp Extensibility Feature)  
 self.\_callbacks = callbacks  
 self.\_helpers = callbacks.getHelpers()  
 # set our extension name  
 callbacks.setExtensionName("Unique Parameter Values")  
 callbacks.registerContextMenuFactory(self)  
 return  
  
  
 def createMenuItems(self, ctxMenuInvocation):  
 self.ctxMenuInvocation = ctxMenuInvocation  
  
 return [self.menuItem]

1. 实现我们的自定义功能UniqueParamValues模块。

把这个文件放在第一步提到的目录里，这个是我们的自定义功能。任何改动都只需要在这个文件里改动，这样就不需要每次都重写导入一次burp扩展了。

Burp插件导入第二部的文件，逻辑实现在这个文件。

def getUniqueParams(self):  
 # Initialize list  
 parameter\_array = []  
 parameter\_string\_array = []  
 messages = self.ctxMenuInvocation.getSelectedMessages()  
 # This for loop iterates through all of the selected messages pulling out   
 # everything Burp considers a parameter (even cookies), and putting all of   
 # the parameters in an array  
 for m in messages:  
 request\_byte\_array = m.getRequest()  
 requestInfo = self.\_helpers.analyzeRequest(request\_byte\_array)  
 parameters = requestInfo.getParameters()  
 parameter\_array = parameter\_array + parameters  
  
 # This for loop iterates through each paramter and creates a string with the  
 # paramname=paramvalue, so that they can be compared and sorted later.  
 for p in parameter\_array:  
 param\_string = p.getName() + "=" + p.getValue()  
 # print "Param String:", param\_string  
 parameter\_string\_array.append(param\_string)  
  
 # After the for loop is finished, then uniquify and sort the parameters -- The main purpose of the extension  
 unique\_parameters = sorted(uniqify(self, parameter\_string\_array))  
  
 print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Unique Paramters \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Unique Paramters \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 print  
 print "Number of Parameters:", len(parameter\_string\_array)  
 print "Number of Unique Parameters :", len(unique\_parameters)  
 print  
  
 param\_dict = {}  
 for unique\_param in unique\_parameters:  
 # print "Param: %s" % (unique\_param))  
 param\_name = unique\_param.split("=")[0]  
 param\_value = unique\_param.split("=")[1]  
 # This if statement creates a dictionary, but unlike a normal dictionary, the value of each key is a list.  
 # This is so that I can use the append function.  
 # The key is the parameter name  
 # The value is a list of all of unique the seen parameter values  
 if not param\_name in param\_dict:  
 param\_dict[param\_name] = []  
 param\_dict[param\_name].append(param\_value)  
  
 for key, value in param\_dict.iteritems():  
 print(len(key) \* "-" + "----")  
 print("| %s |" % (key))  
 print(len(key) \* "-" + "----")  
 for item in value:  
 print(item)  
 print("\n\n\n\n")  
  
  
def uniqify(self, parameter\_string\_array):  
 # not order preserving  
 set = {}  
 map(set.\_\_setitem\_\_, parameter\_string\_array, [])  
 return set.keys()