# Fuzz analysis

Fuzz analysis is a software-testing technique used by auditors and developers to test their applications against unexpected, invalid, and random sets of data input. The response will then be noticed in terms of an exception or a crash thrown by these applications. This activity uncovers some of the major vulnerabilities in the software, which are not possible to discover otherwise. These include buffer overflows, format strings, code injections, dangling pointers, race conditions, denial of service conditions, and many other types of vulnerabilities.

There are different classes of fuzzers available in Kali Linux, which can be used to test the file formats, network protocols, command-line inputs, environmental variables, and web applications. Any untrusted source of data input is considered to be insecure and inconsistent. For instance, a trust boundary between the application and the Internet user is unpredictable. Thus, all the data inputs should be fuzzed and verified against known and unknown vulnerabilities. Fuzzy analysis is a relatively simple and effective solution that can be incorporated into the quality assurance and security testing processes. For this reason, fuzzy analysis is also called robustness testing or negative testing sometimes.

### Note

**What key steps are involved in fuzzy analysis?**

Six common steps should be undertaken. They include identifying the target, identifying inputs, generating fuzz data, executing fuzz data, monitoring the output, and determining the exploitability. These steps are explained in more detail in the *Fuzzing: Brute Force Vulnerability Discovery* presentation available at http://recon.cx/en/f/msutton-fuzzing.ppt.

## BED

**Bruteforce Exploit Detector** (**BED**) is a powerful tool designed to fuzz the plain text protocols against potential buffer overflows, format string bugs, integer overflows, DoS conditions, and so on. It automatically tests the implementation of a chosen protocol by sending different combinations of commands with problematic strings to confuse the target. The protocols supported by this tool are `ftp`, `smtp`, `pop`, `http`, `irc`, `imap`, `pjl`, `lpd`, `finger`, `socks4`, and `socks5`.

To start BED, navigate to **Kali Linux** | **Vulnerability Analysis** | **Fuzzing Tools** | **bed** or use the following command to execute it from your shell:

```
# cd /usr/share/bed/
# bed.pl
```

The usage instructions will now appear on the screen. Note that the description about the specific protocol plugin can be retrieved with the following command:

```
# bed -s FTP
```

In the preceding example, we have successfully learned about the parameters that are required by the FTP plugin before the test execution. These include the FTP `-u` username and `-v` password. Hence, we have demonstrated a small test against our target system running the FTP daemon.

```
# bed -s FTP -u ftpuser -v ftpuser -t 192.168.0.7 -p 21 -o 3

BED 0.5 by mjm ( www.codito.de ) & eric ( www.snake-basket.de)

 + Buffer overflow testing:
               testing: 1      USER XAXAX      ..........
               testing: 2      USER ftpuserPASS XAXAX  ..........
 + Formatstring testing:
               testing: 1      USER XAXAX      .......
               testing: 2      USER ftpuserPASS XAXAX  .......
* Normal tests
 + Buffer overflow testing:
               testing: 1      ACCT XAXAX      ..........
               testing: 2      APPE XAXAX      ..........
               testing: 3      ALLO XAXAX      ..........
               testing: 4      CWD XAXAX       ..........
```

```
                    testing: 5      CEL XAXAX       ..........
                    testing: 6      DELE XAXAX      ..........
                    testing: 7      HELP XAXAX      ..........
                    testing: 8      MDTM XAXAX      ..........
                    testing: 9      MLST XAXAX      ..........
                    testing: 10     MODE XAXAX      ..........
                    testing: 11     MKD XAXAX       ..........
                    testing: 12     MKD XAXAXCWD XAXAX      ..........
                    testing: 13     MKD XAXAXDELE XAXAX     ..........
                    testing: 14     MKD XAXAXRMD XAXAX      .....connection
attempt failed: No route to host
```

From the output, we can anticipate that the remote FTP daemon has been interrupted during the fourteenth test case. This could be a clear indication of a buffer overflow bug; however, the problem can be further investigated by looking into a specific plugin module and locating the pattern of the test case (for example, `/usr/share/bed/bedmod/ftp.pm`). It is always a good idea to test your target at least two more times by resetting it to a normal state, increasing the timeout value (`-o`), and checking if the problem is reproducible.

## JBroFuzz

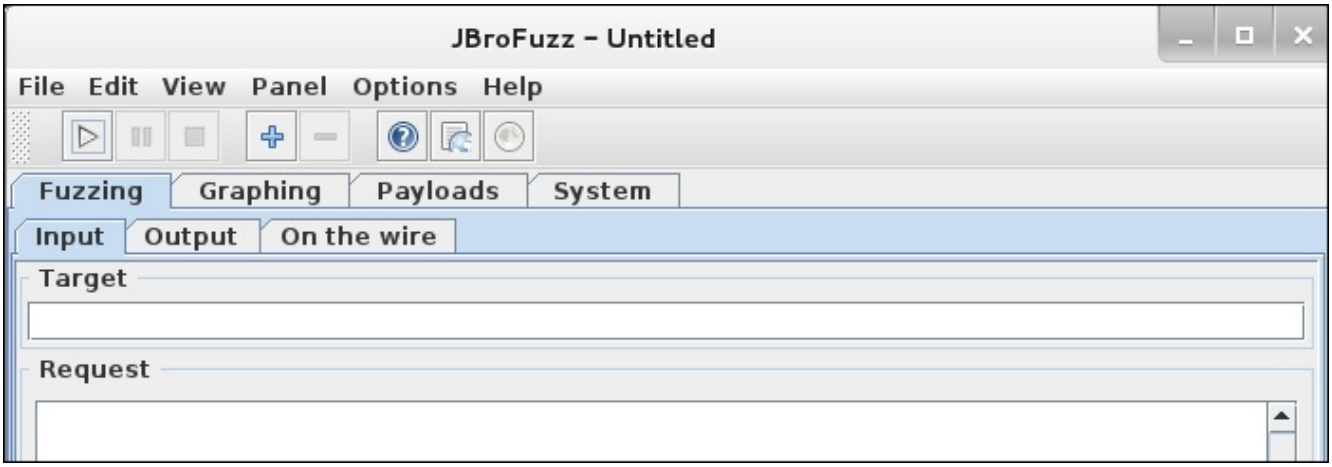
JBroFuzz is a well-known platform to fuzzy test web applications. It supports web requests over the HTTP and HTTPS protocol. By providing a simple URL for the target domain and selecting the part of a web request to fuzz, an auditor can either select to craft the manual request or use the predefined set of payloads database (for example, cross-site scripting, SQL injection, buffer overflow, format string errors, and so on) to generate some malicious requests based on the previously known vulnerabilities and send them to the target web server. The corresponding responses will then be recorded for further inspection. Based on the type of testing that is performed, these responses or results should be manually investigated in order to recognize any possible exploit condition.

The key options provided under JBroFuzz are fuzz management, payload categories, sniffing the web requests and replies through browser proxy, and enumerating the web directories. Each of these has unique functions and capabilities to handle application protocol fuzzing.

To start JBroFuzz, use the console to execute the following commands:

```
# cd /usr/share/zaproxy/lib/jbrofuzz/
# java -jar JBroFuzz.jar
```

Once the GUI application is loaded, you can visit a number of available options to learn more about their prospects. If you need any assistance, go to the menu bar and navigate to **Help** | **Topics**, as shown in the following screenshot:




Now let's take an example by testing the target web application using the following steps:

1. We select the URL of our target domain as `http://testasp.example.com`, which hosts the ASP web application. In the **Request** panel, we also modify the HTTP request to suit our testing criteria as follows:

```
GET /showthread.asp?id=4 HTTP/1.0
Host: testasp.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-GB;
rv:1.9.0.10) Gecko/2009042316 Firefox/3.0.10
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*
```

```
/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

2. Before crafting the preceding request, we already knew that the resource URL, `http://testasp.example.com/showthread.asp?id=4`, does exist on the web server.
3. Create a manual request and then target the specific part of a URL ( `id=4` ) with a SQL injection payload.
4. Highlight a numeric value, `4`, in the first line and click on the add button (**+**) in the top toolbar.
5. In the new window, select the **SQL Injection** category, fuzzer name **SQL Injection**, and click on the **Add Fuzzer** button.
6. Once the fuzzer is finalized, you will see that it is listed under **Added Payloads Table** in the right-hand corner of the main window.

If you followed the preceding steps thoroughly, you are now ready to start fuzzing the target web application against a set of SQL injection vulnerabilities.

To start, go to the menu bar and navigate to **Panel** | **Start**, or use the *Ctrl* + *Enter* shortcut from your keyboard. As the requests are getting processed, you will see that the output has been logged in the table below the **Request** panel. Additionally, you may be interested in catching up on the progress of each HTTP(s) request that can be done through the use of the **On The Wire** tab. After the fuzzy session is complete, you can investigate each response based on the crafted request. This can be done by clicking on the specific response in the **Output** window and right-clicking on it to choose the **Open in Browser** option. We got the following response for one of our requests, which clearly shows us the possibility of a SQL injection vulnerability:

```
HTTP/1.1 500 Internal Server Error Connection: close Date: Sat, 04 Sep
2013 21:59:06 GMT Server: Microsoft-IIS/6.0 X-Powered-By: ASP.NET Content-
Length: 302 Content-Type: text/html Set-Cookie:
ASPSESSIONIDQADTCRCB=KBLKHENAJBNNKIOKKAJJFCDI; path=/ Cache-control:
private

Microsoft SQL Native Client error '80040e14'
Unclosed quotation mark after the character string ''.
/showthread.asp, line 9
```

**Note**

For more information, visit http://wiki191.owasp.org/index.php/Category:OWASP_JBroFuzz.