Using Burp to Detect SQL Injection Via SQL-Specific Parameter Manipulation

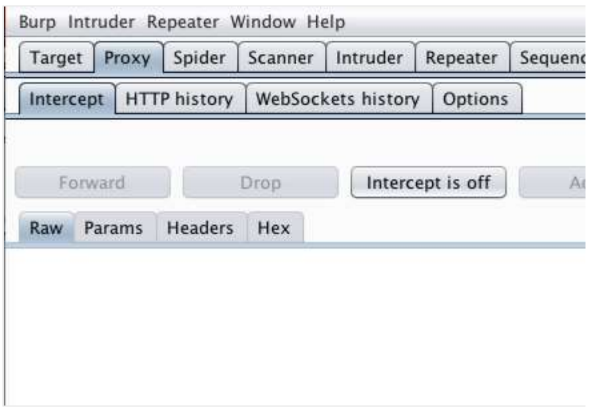
In the most obvious cases, a SQL injection flaw may be discovered and conclusively verified by supplying a single item of unexpected input to the applic other cases, bugs may be extremely subtle and may be difficult to distinguish from other categories of vulnerability or from benign anomalies that do not 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 obs the effects on application responses.

Manual testing for numeric SQL-specific parameter manipulation

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

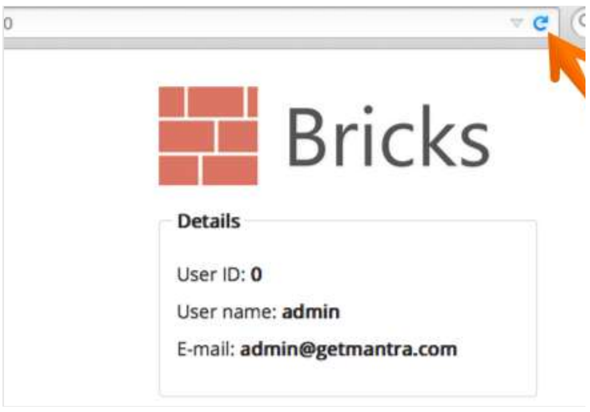
Ensure "Intercept is off" in the Proxy "Intercept" tab.



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.



Details

User ID: **0**

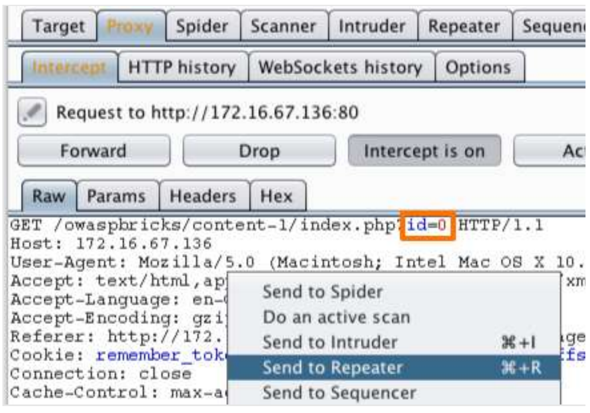
User name: **admin**

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

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".



```
GET /owaspbricks/content-1/index.php?id=0 HTTP/1.1
Host: 172.16.67.136
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.
Accept: text/html,ap                                    xm
Accept-Language: en-
Accept-Encoding: gzi
Referer: http://172.                                    ge
Cookie: remember_tok                                    fs
Connection: close
Cache-Control: max-a
```

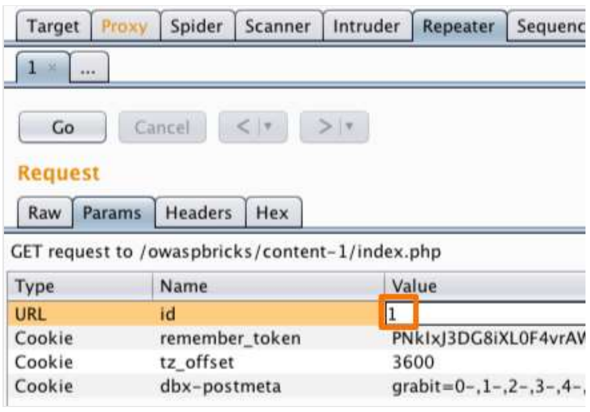
Go to the Repeater tab.

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.



GET request to /owaspbricks/content-1/index.php

Type	Name	Value
URL	id	1
Cookie	remember_token	PNkIxJ3DG8iXL0F4vrAV
Cookie	tz_offset	3600
Cookie	dbx-postmeta	grabit=0-,1-,2-,3-,4-,

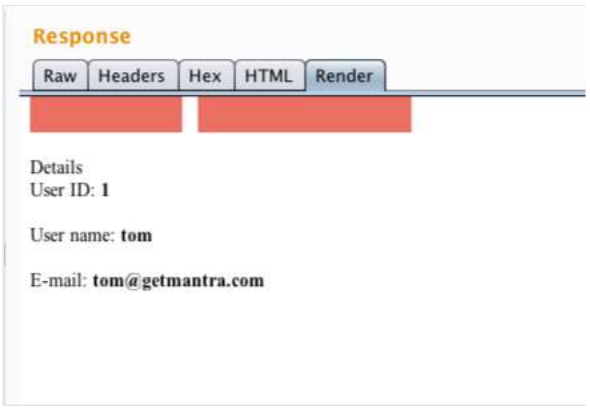
The results of the request can be viewed in the response section of the "Repeater" tool.

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.



Details
User ID: **1**

User name: **tom**

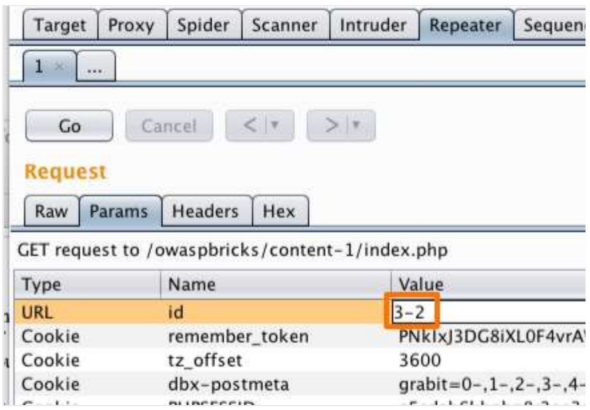
E-mail: **tom@getmantra.com**

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.



GET request to /owaspbricks/content-1/index.php

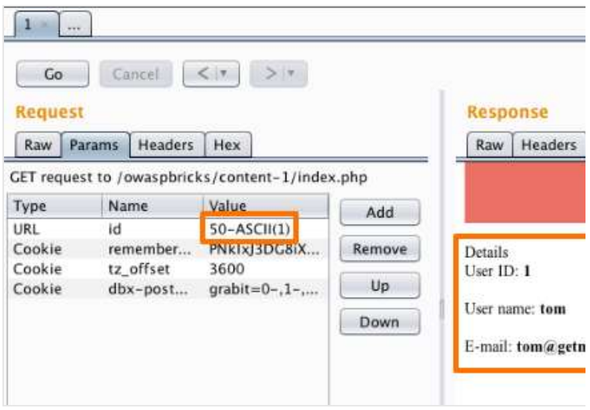
Type	Name	Value
URL	id	3-2
Cookie	remember_token	PNkIxJ3DG8iXL0F4vrA
Cookie	tz_offset	3600
Cookie	dbx-postmeta	grabit=0-,1-,2-,3-,4-

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.



GET request to /owaspbricks/content-1/index.php

Type	Name	Value
URL	id	50-ASCII(1)
Cookie	remember...	PNkIxJ3DG8iX...
Cookie	tz_offset	3600
Cookie	dbx-post...	grabit=0-,1-,...

Details
User ID: **1**

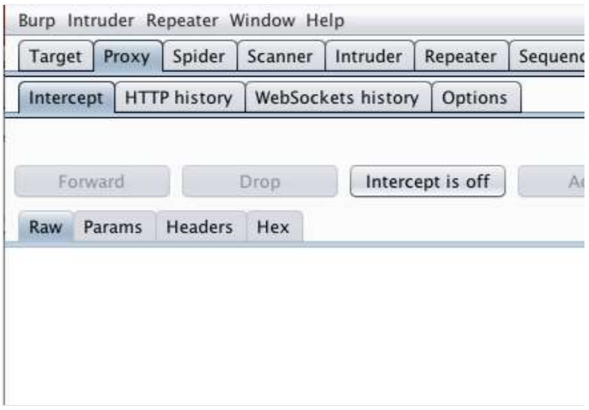
User name: **tom**

E-mail: **tom@getn**

Manual testing for string based SQL-specific parameter manipulation

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

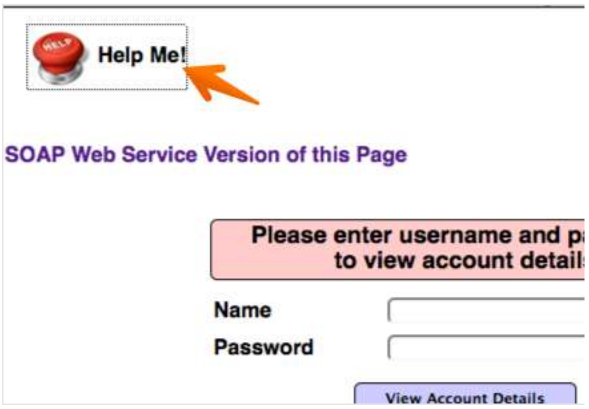
Ensure "Intercept is off" in the Proxy "Intercept" tab.


Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequenc
Intercept HTTP history WebSockets history Options
Forward Drop Intercept is off
Raw Params Headers Hex

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

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

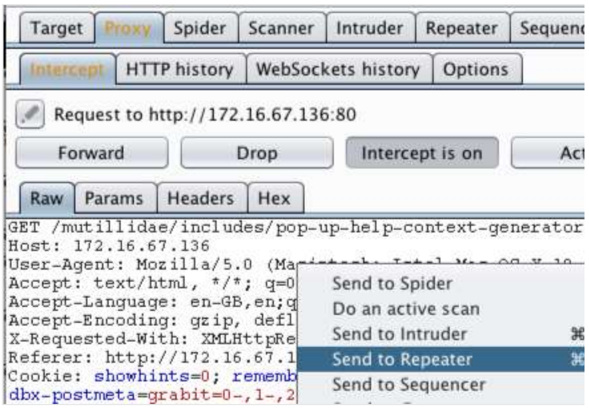
Now send a request to the server. In this example by clicking the help button.



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".


Target Proxy Spider Scanner Intruder Repeater Sequenc
Intercept HTTP history WebSockets history Options
Forward Drop Intercept is on Act
Raw Params Headers Hex
Send to Spider
Do an active scan
Send to Intruder
Send to Repeater
Send to Sequencer

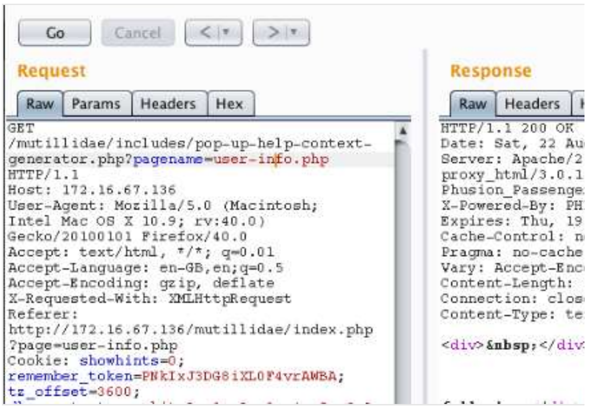
Request to http://172.16.67.136:80

```
GET /mutillidae/includes/pop-up-help-context-generator
Host: 172.16.67.136
User-Agent: Mozilla/5.0 (Ma
Accept: text/html, */*; q=0
Accept-Language: en-GB,en;q
Accept-Encoding: gzip, defl
X-Requested-With: XMLHttpRe
Referer: http://172.16.67.1
Cookie: showhints=0; rememb
dbx-postmeta=grabit=0-,1-,2
```

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.



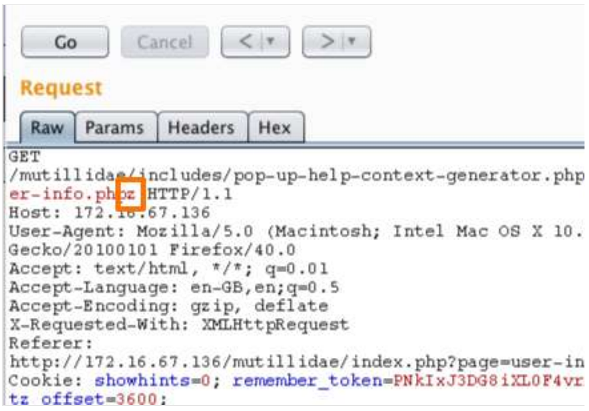
```
GET
/mutillidae/includes/pop-up-help-context-
generator.php?pagename=user-info.php
HTTP/1.1
Host: 172.16.67.136
User-Agent: Mozilla/5.0 (Macintosh;
Intel Mac OS X 10.9; rv:40.0)
Gecko/20100101 Firefox/40.0
Accept: text/html, */*; q=0.01
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Referer:
http://172.16.67.136/mutillidae/index.php
?page=user-info.php
Cookie: showhints=0;
remember_token=PNkIxJ3DG8iXL0F4vrAWBA;
tz_offset=3600;
```

```
HTTP/1.1 200 OK
Date: Sat, 22 Au
Server: Apache/2
proxy_html/3.0.1
Phusion_Passenge
X-Powered-By: PH
Expires: Thu, 19
Cache-Control: n
Pragma: no-cache
Vary: Accept-Enc
Content-Length:
Connection: clos
Content-Type: te

<div>&nbsp;</div
```

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

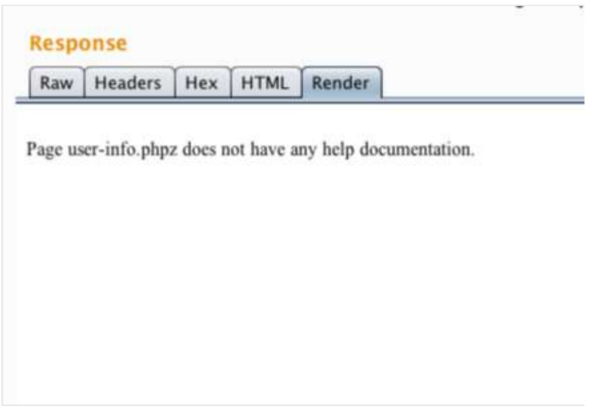
In this example we have added a single letter "z" to the value of the parameter.



```
GET
/mutillidae/includes/pop-up-help-context-generator.php
er-info.phpz HTTP/1.1
Host: 172.16.67.136
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.
Gecko/20100101 Firefox/40.0
Accept: text/html, */*; q=0.01
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Referer:
http://172.16.67.136/mutillidae/index.php?page=user-in
Cookie: showhints=0; remember_token=PNkIxJ3DG8iXL0F4vr
tz_offset=3600;
```

The additional letter causes a change in the response from the application.

This demonstrates that the parameter is being used to retrieve content.



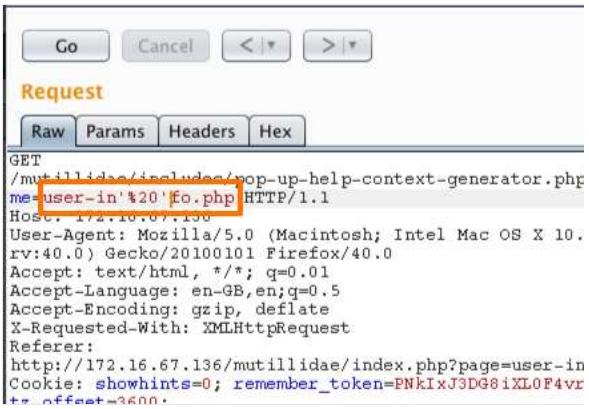
Page user-info.phpz does not have any help documentation.

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.

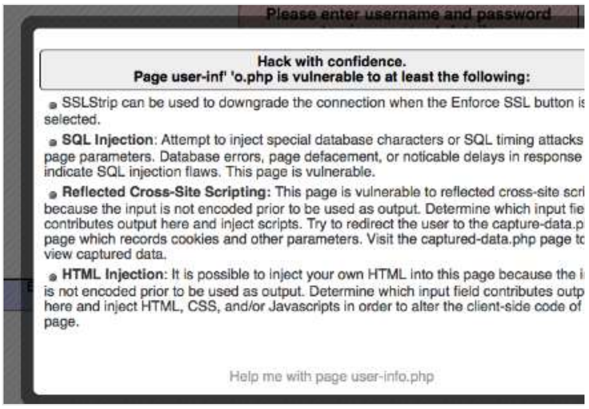


```
GET
/mutillidae/includes/pop-up-help-context-generator.php
me=user-in'%20'fo.php HTTP/1.1
Host: 172.16.67.136
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.
rv:40.0) Gecko/20100101 Firefox/40.0
Accept: text/html, */*; q=0.01
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Referer:
http://172.16.67.136/mutillidae/index.php?page=user-in
Cookie: showhints=0; remember_token=PNkIxJ3DG8iXL0F4vr
tz_offset=3600;
```

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.

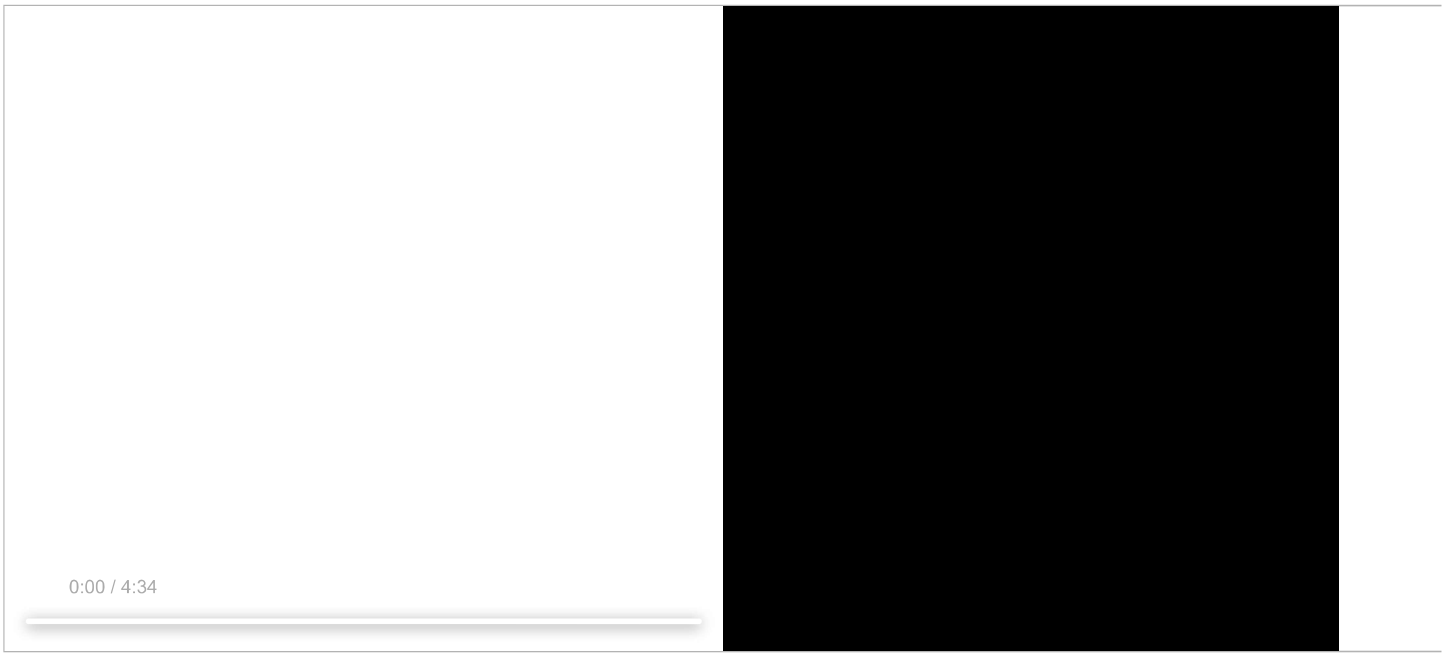
We can verify this in the "Response" section of the repeater tab or by sending the request to the application via the browser.



Hack with confidence.
Page user-inf' 'o.php is vulnerable to at least the following:

- SSLStrip can be used to downgrade the connection when the Enforce SSL button is selected.
- **SQL Injection**: Attempt to inject special database characters or SQL timing attacks page parameters. Database errors, page defacement, or noticable delays in response indicate SQL injection flaws. This page is vulnerable.
- **Reflected Cross-Site Scripting:** This page is vulnerable to reflected cross-site scri because the input is not encoded prior to be used as output. Determine which input fie contributes output here and inject scripts. Try to redirect the user to the capture-data.p page which records cookies and other parameters. Visit the captured-data.php page to view captured data.
- **HTML Injection**: It is possible to inject your own HTML into this page because the i is not encoded prior to be used as output. Determine which input field contributes outp here and inject HTML, CSS, and/or Javascripts in order to alter the client-side code of page.

Help me with page user-info.php



Related articles:

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

Burp Suite
Web vulnerability scanner
Burp Suite Editions
Release Notes

Vulnerabilities
Cross-site scripting (XSS)
SQL injection
Cross-site request forgery
XML external entity injection
Directory traversal
Server-side request forgery

Customers
Organizations
Testers
Developers

Company
About
PortSwigger News
Careers
Contact
Legal
Privacy Notice

Insights
Web Security Academy
Blog
Research
The Daily Swig

Follow us