**Server-Side Parameter Pollution:**

**VERY Useful article here:** [**https://owasp.org/www-project-web-security-testing-guide/latest/4-Web\_Application\_Security\_Testing/07-Input\_Validation\_Testing/04-Testing\_for\_HTTP\_Parameter\_Pollution**](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/04-Testing_for_HTTP_Parameter_Pollution)

**If running Arjun for finding hidden inputs we can always dwonlaod the param miner wordlists to be used in arjun heres where they can be found:** [**https://github.com/PortSwigger/param-miner/tree/master/resources**](https://github.com/PortSwigger/param-miner/tree/master/resources)

**Also try running param miner with other wordlists (or just run arjun) good word lists for parasm are seclists params (saved in lists and payloads on windows or in seclists on kali), samslist on kali, or arjun word lists**

**\*\*\*\* PARAM miner issues tons of requests and burp does not allow rate limits by default. ARJUN does allow for delays between requests.Arjun supports GET/POST/POST-JSON/POST-XML requests. It also automatically handles rate limits or timeouts byt we can manually delay. It also has functionality to extract params from JS or external sources.** **https://github.com/s0md3v/Arjun/wiki/Usage**

Some systems contain internal APIs that aren't directly accessible from the internet. Server-side parameter pollution occurs when a website embeds user input in a server-side request to an internal API without adequate encoding. This means that an attacker may be able to manipulate or inject parameters, which may enable them to, for example:

* Override existing parameters.
* Modify the application behavior.
* Access unauthorized data.

You can test any user input for any kind of parameter pollution. For example, query parameters, form fields, headers, and URL path parameters may all be vulnerable.

\*\* this vulnerability is sometimes called HTTP param pollution but that term also refers to WAF bypass technique which is VERY different from this so well call it server-side param pollution.\*\*

**Testing For Param Pollution in Query String:**

To test for server-side parameter pollution in the query string, place query syntax characters like #, &, and = in your input and observe how the application responds.

Consider a vulnerable application that enables you to search for other users based on their username. When you search for a user, your browser makes the following request:

GET /userSearch?name=peter&back=/home

To retrieve user information, the server queries an internal API with the following request:

GET /users/search?name=peter&publicProfile=true

**Truncating query strings**

You can use a URL-encoded # character to attempt to truncate the server-side request. To help you interpret the response, you could also add a string after the # character.

For example, you could modify the query string to the following:

GET /userSearch?name=peter%23foo&back=/home

The front-end will try to access the following URL:

GET /users/search?name=peter#foo&publicProfile=true

**\*\*It's essential that you URL-encode the # character. Otherwise the front-end application will interpret it as a fragment identifier and it won't be passed to the internal API.\*\***

Review the response for clues about whether the query has been truncated. For example, if the response returns the user peter, the server-side query may have been truncated. If an Invalid name error message is returned, the application may have treated foo as part of the username. This suggests that the server-side request may not have been truncated.

If you're able to truncate the server-side request, this removes the requirement for the publicProfile field to be set to true. You may be able to exploit this to return non-public user profiles.

----------------------------------

**Injecting invalid parameters**

You can use an URL-encoded & character to attempt to add a second parameter to the server-side request.

For example, you could modify the query string to the following:

GET /userSearch?name=peter%26foo=xyz&back=/home

This results in the following server-side request to the internal API:

GET /users/search?name=peter&foo=xyz&publicProfile=true

Review the response for clues about how the additional parameter is parsed. For example, if the response is unchanged this may indicate that the parameter was successfully injected but ignored by the application.

To build up a more complete picture, you'll need to test further.

---------------

**Injecting valid parameters**

If you're able to modify the query string, you can then attempt to add a second valid parameter to the server-side request.

**\*\*\*\* This is where param miner / hidden param tools come into play. Try to identify hidden params that are accepted. Can also be useful to use a tool like arjun or just a fuzzer to fuzz their the body params or query params and look for changes in response. Something like “&fuzzparam=x” could be good because apps often will throw an invalid param value error message or something like that. We can then look for different responses to determine hidden params. \*\*\*\***

For example, if you've identified the email parameter, you could add it to the query string as follows:

GET /userSearch?name=peter%26email=foo&back=/home

This results in the following server-side request to the internal API:

GET /users/search?name=peter&email=foo&publicProfile=true

Review the response for clues about how the additional parameter is parsed.

--------------------------------------------

**Overriding existing parameters**

To confirm whether the application is vulnerable to server-side parameter pollution, you could try to override the original parameter. Do this by injecting a second parameter with the same name.

For example, you could modify the query string to the following:

GET /userSearch?name=peter%26name=carlos&back=/home

This results in the following server-side request to the internal API:

GET /users/search?name=peter&name=carlos&publicProfile=true

The internal API interprets two name parameters. The impact of this depends on how the application processes the second parameter. This varies across different web technologies. For example:

* PHP parses the last parameter only. This would result in a user search for carlos.
* ASP.NET combines both parameters. This would result in a user search for peter,carlos, which might result in an Invalid username error message.
* Node.js / express parses the first parameter only. This would result in a user search for peter, giving an unchanged result.

If you're able to override the original parameter, you may be able to conduct an exploit. For example, you could add name=administrator to the request. This may enable you to log in as the administrator user.

**Test for Server-Side Param Pollution in REST paths:**

A RESTful API may place parameter names and values in the URL path, rather than the query string OR post body. For example, consider the following path:

/api/users/123

The URL path might be broken down as follows:

* /api is the root API endpoint.
* /users represents a resource, in this case users.
* /123represents a parameter, here an identifier for the specific user.

Consider an application that enables you to edit user profiles based on their username. Requests are sent to the following endpoint:

GET /edit\_profile.php?name=peter

This results in the following server-side request:

GET /api/private/users/peter

An attacker may be able to manipulate server-side URL path parameters to exploit the API. To test for this vulnerability, add [path traversal](https://portswigger.net/web-security/file-path-traversal) sequences to modify parameters and observe how the application responds.

You could submit URL-encoded peter/../admin as the value of the name parameter:

GET /edit\_profile.php?name=peter%2f..%2fadmin

This may result in the following server-side request:

GET /api/private/users/peter/../admin

If the server-side client or back-end API normalize this path, it may be resolved to /api/private/users/admin.

**\*\*\*\* Essentially if a certain API path is blocked consider path traversal attacks to try to access previously blocked paths or endpoints \*\*\*\***

**Testing for server-side parameter pollution in structured data formats:**

**\*\* this involves finding hidden params and injecting them in the body of a standard post request. We do not always have to put the data in json format in the request. Again just use param miner or Arjun to find hidden params in the query string or request body. \*\***

An attacker may be able to manipulate parameters to exploit vulnerabilities in the server's processing of other structured data formats, such as a JSON or XML. To test for this, inject unexpected structured data into user inputs and see how the server responds.

Consider an application that enables users to edit their profile, then applies their changes with a request to a server-side API. When you edit your name, your browser makes the following request:

POST /myaccount

name=peter

This results in the following server-side request:

PATCH /users/7312/update

{"name":"peter"}

You can attempt to add the access\_level parameter to the request as follows:

POST /myaccount

name=peter","access\_level":"administrator

If the user input is added to the server-side JSON data without adequate validation or sanitization, this results in the following server-side request:

PATCH /users/7312/update

{name="peter","access\_level":"administrator"}

This may result in the user peter being given administrator access.

-----------

Consider a similar example, but where the client-side user input is in JSON data. When you edit your name, your browser makes the following request:

POST /myaccount

{"name": "peter"}

This results in the following server-side request:

PATCH /users/7312/update

{"name":"peter"}

You can attempt to add the access\_level parameter to the request as follows:

POST /myaccount

{"name": "peter\",\"access\_level\":\"administrator"}

If the user input is decoded, then added to the server-side JSON data without adequate encoding, this results in the following server-side request:

PATCH /users/7312/update

{"name":"peter","access\_level":"administrator"}

Again, this may result in the user peter being given administrator access.

Structured format injection can also occur in responses. For example, this can occur if user input is stored securely in a database, then embedded into a JSON response from a back-end API without adequate encoding. You can usually detect and exploit structured format injection in responses in the same way you can in requests.

**Note**

This example below is in JSON, but server-side parameter pollution can occur in any structured data format. For an example in XML, see the [XInclude attacks](https://portswigger.net/web-security/xxe" \l "xinclude-attacks) section in the XML external entity ([XXE](https://portswigger.net/web-security/xxe)) injection topic.

**Testing W/ Automated Tools:**

Burp scanner is whats recommended. Just use arjun/param miner to find hidden inputs and try to inject values into them. **Also try SQLi in some interesting parameters.**

**Extra but Useful Info:**

Be sure to try path traversal when trying to access other users. So like /api/v1/user/kole/../admin essentially your username or user id followed by ../ or ./ followed by anothers users name or user id.

------------------

We can also try to put common api file names in the url/ query string and look for error messages that may contain a path like this:

add some common API definition filenames to the URL path. For example, submit the following:

username=../../../../openapi.json%23

Notice that this returns an error message, which contains the following API endpoint for finding users:

/api/internal/v1/users/{username}/field/{field}

-------------------

Also try to change the api version to potentially access new attack surfaces or vulnerabilities