



Discovering Vulnerabilities With Nuclei

Author: <https://github.com/small-goblin>

This is a quick introduction to the tool nuclei. Nuclei is a DAST scanner, or dynamic application scanning tool. DAST is a complex topic; the TLDR is that it can enable researchers and testers to

- discover new vulnerabilities
- test for existing, known vulnerabilities (like CVEs)
- and provide a mechanism to make vulnerabilities re-testable to other technical teams

Nuclei really shines with its supportive community, scanning speed, and robust template engine.

You should

- already have some familiarity with Linux, and the terminal
- understand the gist of common vulnerabilities for web applications, like SQL injection

If not, check out:

- <https://ubuntu.com/tutorials/command-line-for-beginners#1-overview>
- https://www.w3schools.com/sql/sql_injection.asp

Lab

For much of this lab, you will be using the terminal.

The terminal is this thing, on the far left:



1. Start Juice Shop

Terminal: sudo apt install juice-shop

Terminal: sudo juice-shop

```
user@kali:~$ 
└$ 
  (user@kali)-[~] 
  $ 

  (user@kali)-[~] 
  $ sudo juice-shop
[*] Please wait for the Juice-shop service to start.
[*]
[*] You might need to refresh your browser once it opens.
[*]
[*] Web UI: http://127.0.0.1:42000

● juice-shop.service - juice-shop web application
   Loaded: loaded (/usr/lib/systemd/system/juice-shop.service; disabled; preset: disabled)
   Active: active (running) since Fri 2025-10-24 19:32:06 EDT; 17s ago
     Invocation: 277bf90107fe423290f4e11dd62031a1
      Main PID: 2664 (npm start)
        Tasks: 23 (limit: 4414)
       Memory: 233.8M (peak: 233.8M)
         CPU: 11.799s
        CGroup: /system.slice/juice-shop.service
                  └─2664 "npm start"
                    ├─2707 sh -c "node build/app"
                    └─2708 node build/app

Oct 24 19:32:06 kali systemd[1]: Started juice-shop.service - juice-shop w...tion.
Oct 24 19:32:10 kali npm[2664]: > juice-shop@16.0.1 start
Oct 24 19:32:10 kali npm[2664]: > node build/app
Oct 24 19:32:14 kali npm[2708]: info: All dependencies in ./package.json a... (OK)
Oct 24 19:32:23 kali npm[2708]: info: Detected Node.js version v20.19.5 (OK)
Oct 24 19:32:23 kali npm[2708]: info: Detected OS linux (OK)
Oct 24 19:32:23 kali npm[2708]: info: Detected CPU x64 (OK)
Oct 24 19:32:23 kali npm[2708]: info: Configuration default validated (OK)
Hint: Some lines were ellipsized, use -l to show in full.

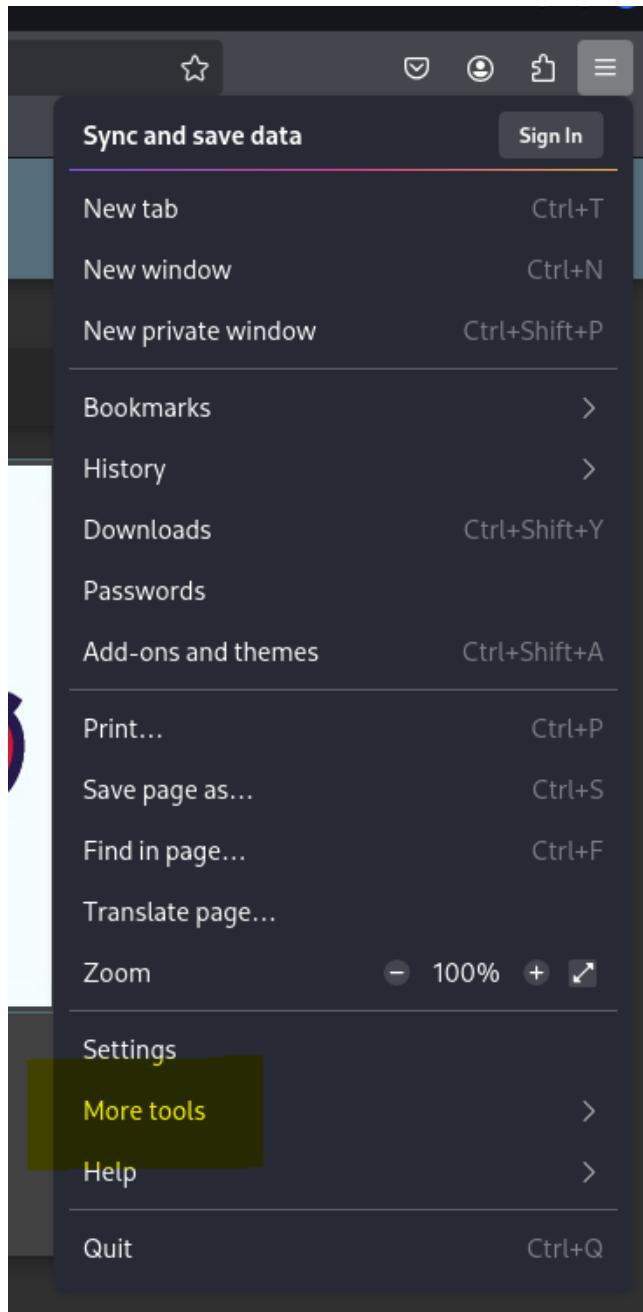
[*] Opening Web UI (http://127.0.0.1:42000) in: 5... 4... 3... 2... 1...

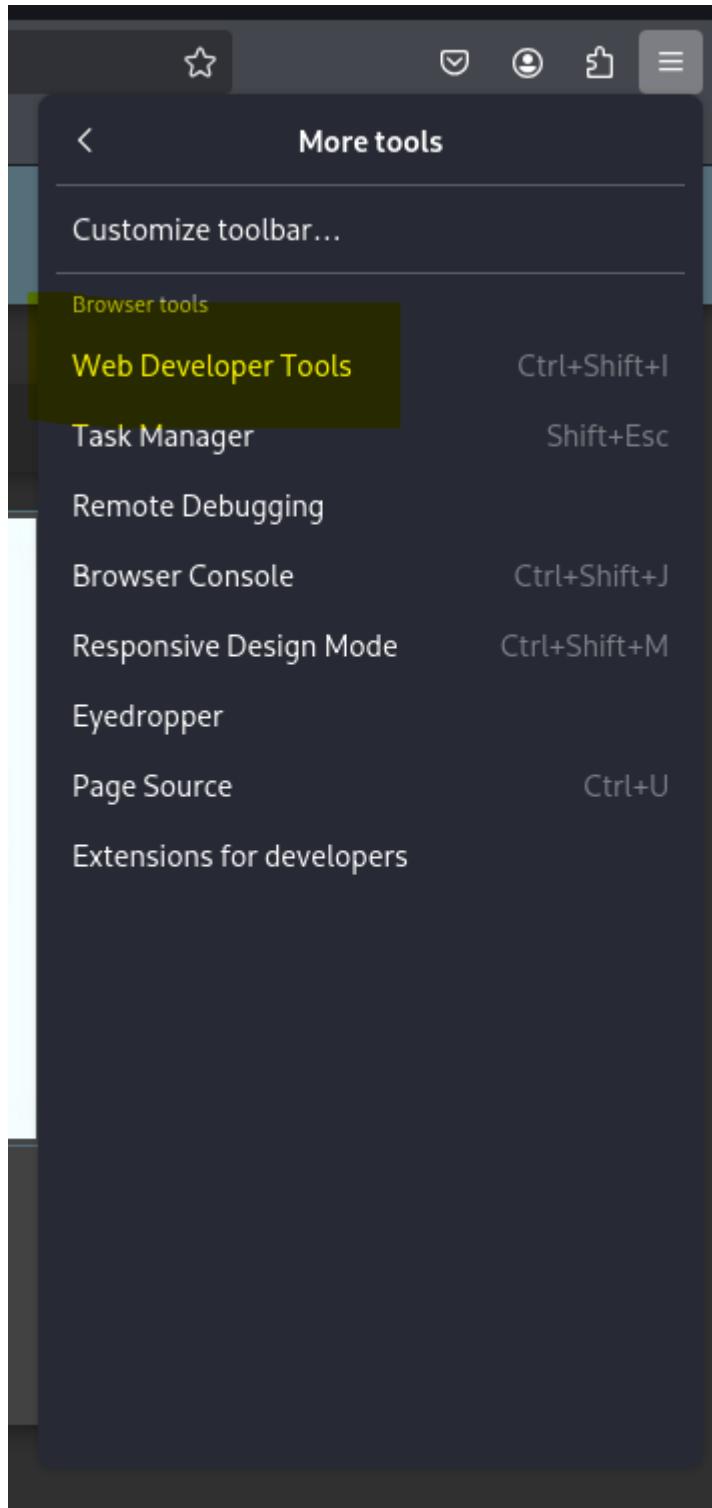
  (user@kali)-[~] 
  $ 
```

A web browser should automatically open.

If not, the default address is: <http://127.0.0.1:4200>

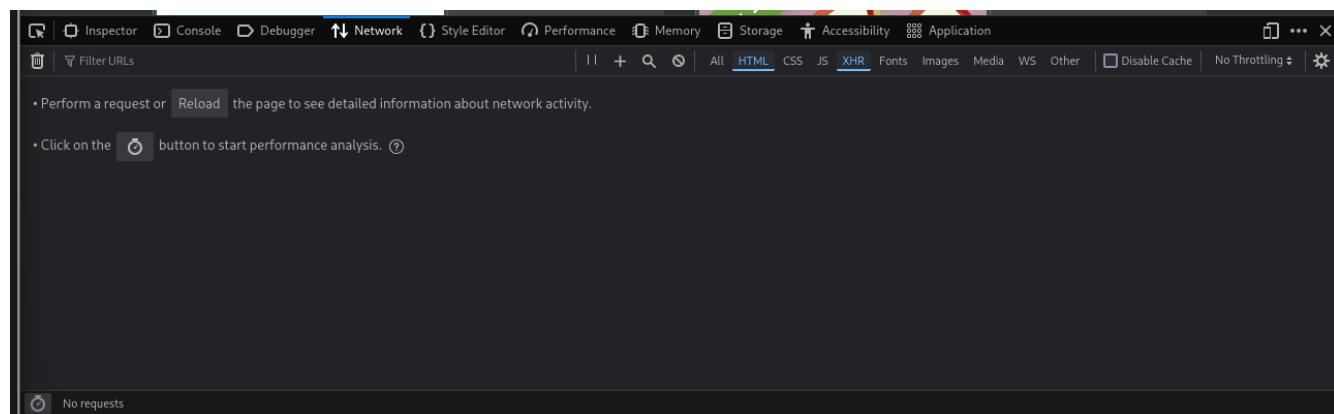
2. Open up Firefox Web Developer Tools.





Or Ctrl Shift I

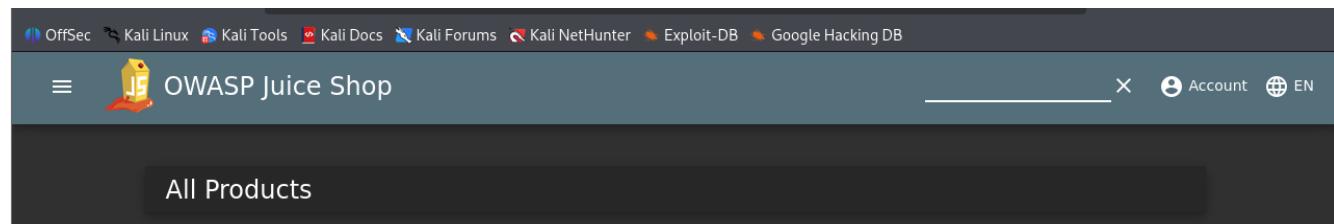
Select the “Network” tab



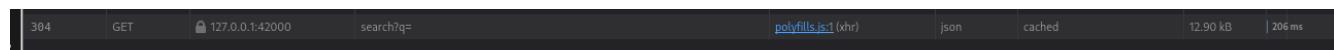
Before testing, normally, you would first explore an application and map out the entire API, cookies, parameters, etc.

We're going to keep an eye out for interesting traffic.

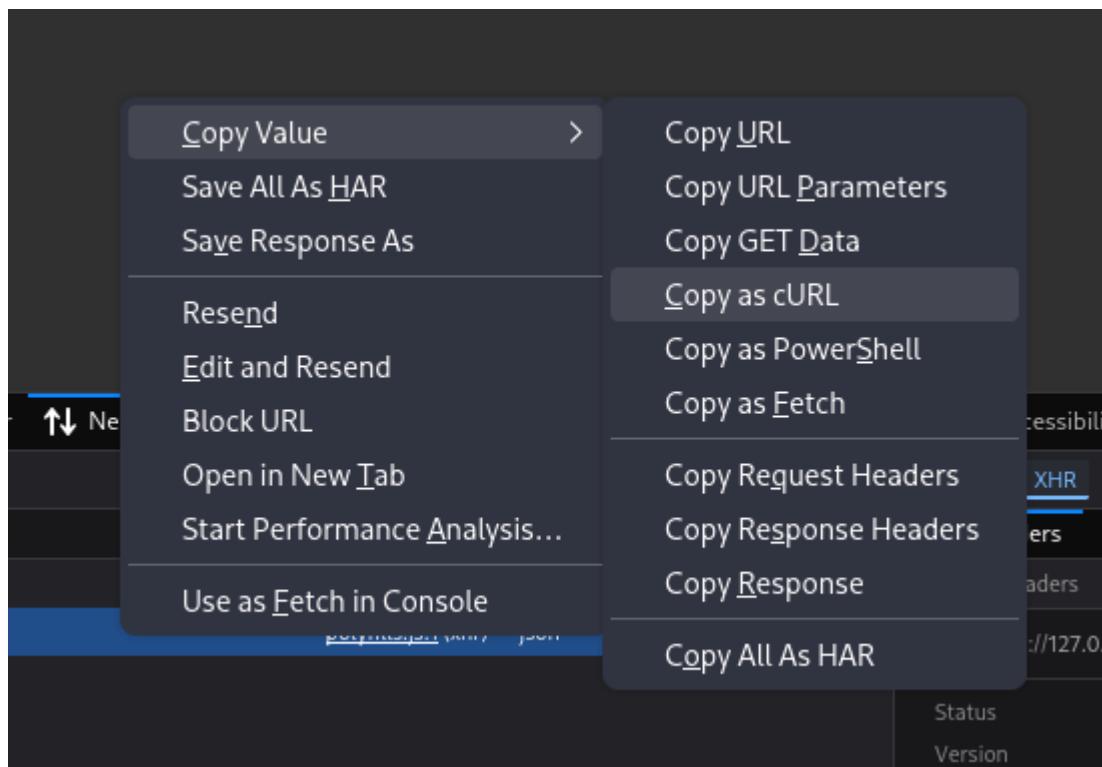
Lets search for products



And explore the resulting request



Right click on the request, “copy as cURL”



Paste in a Text Editor

```
curl http://127.0.0.1:42000
Draft
```

```
1 curl 'http://127.0.0.1:42000/rest/products/search?q=' \
2   -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0' \
3   -H 'Accept: application/json, text/plain, */*' \
4   -H 'Accept-Language: en-US,en;q=0.5' \
5   -H 'Accept-Encoding: gzip, deflate, br, zstd' \
6   -H 'X-User-Email: test@test.com' \
7   -H 'Connection: keep-alive' \
8   -H 'Referer: http://127.0.0.1:42000/' \
9   -H 'Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss' \
10  -H 'Sec-Fetch-Dest: empty' \
11  -H 'Sec-Fetch-Mode: cors' \
12  -H 'Sec-Fetch-Site: same-origin' \
13  -H 'If-None-Match: W/"325f-uH/N1p+g3B3lxfTnZiUpn72eObQ"' \
14  -H 'Priority: u=0'
```

The terminal window shows a curl command with 14 lines of headers. The command is intended to search for products with the query parameter 'q='.

If we were to run this in the terminal, it would return nothing. Likely, “search?q=” is a parameter for the term we are searching on, and searching on nothing returns nothing.

If we change this to “search?q=apple” we get data about products that match that search term.

```
(user㉿kali)-[~]
$ curl 'http://127.0.0.1:42000/rest/products/search?q=apple' \
-H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0' \
-H 'Accept: application/json, text/plain, */*' \
-H 'Accept-Language: en-US,en;q=0.5' \
-H 'Accept-Encoding: gzip, deflate, br, zstd' \
-H 'X-User-Email: test@test.com' \
-H 'Connection: keep-alive' \
-H 'Referer: http://127.0.0.1:42000/' \
-H 'Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss' \
-H 'Sec-Fetch-Dest: empty' \
-H 'Sec-Fetch-Mode: cors' \
-H 'Sec-Fetch-Site: same-origin' \
-H 'If-None-Match: W/"325f-uH/N1p+g3B3lxfTnZiUpn72e0bQ"' \
-H 'Priority: u=0'
{"status": "success", "data": [{"id": 1, "name": "Apple Juice (1000ml)", "description": "The all-time classic.", "price": 1.99, "deluxePrice": 0.99, "image": "apple_juice.jpg", "createdAt": "2025-10-24 23:32:27.829 +00:00", "updatedAt": "2025-10-24 23:32:27.829 +00:00", "deletedAt": null}, {"id": 24, "name": "Apple Pomace", "description": "Finest pressings of apples. Allergy disclaimer: Might contain traces of worms. Can be <a href=\"/#recycle\">sent back to us</a> for recycling.", "price": 0.89, "deluxePrice": 0.89, "image": "apple_pressings.jpg", "createdAt": "2025-10-24 23:32:27.835 +00:00", "updatedAt": "2025-10-24 23:32:27.835 +00:00", "deletedAt": null}]}
(user㉿kali)-[~]
$
```

Compare that to the data in the UI

The screenshot shows the OWASP Juice Shop application interface. At the top, there is a logo and the text "OWASP Juice Shop". A search bar contains the text "apple". To the right of the search bar are account-related icons and a close button. Below the header, the text "Search Results - apple" is displayed. Two product items are listed: "Apple Juice (1000ml)" with a price of "1.99¤" and an image of a juice cup and an apple; and "Apple Pomace" with a price of "0.89¤" and an image of a container with apples. At the bottom of the page, there is a footer with pagination controls: "Items per page: 12" with a dropdown arrow, "1 - 2 of 2", and navigation arrows.

We can consider this an “interesting” api call, and scan it using nuclei.

3. First update nuclei template(s)

nuclei -ut

The flags we need to scan a single api endpoint

```
1 nuclei -dast \
2 -u http://127.0.0.1:42000/rest/products/search?q=apple \
3 -fa high|
```

-dast to enable the dynamic scanner

-u to point to our request

-fa high to enable fuzzing aggression to “high”

```
projectdiscovery v3.4.10
[WRN] Found 1 templates with syntax error (use -validate flag for further examination)
[WRN] Found 2 templates with runtime error (use -validate flag for further examination)
[INF] Current nuclei version: v3.4.10 (latest)
[INF] Current nuclei-templates version: v10.3.0 (latest)
[INF] New templates added in latest release: 124
[INF] Templates loaded for current scan: 239
[INF] Executing 239 signed templates from projectdiscovery/nuclei-templates
[INF] Targets loaded for current scan: 1
[sql-error-based:sqlite] [http] [critical] http://127.0.0.1:42000/rest/products/search?q=apple' ["SQLITE_ERROR"] [query:q] [GET]
[INF] Scan completed in 59.019318782s. 1 matches found.

(user@kali)-[~]
$
```

Nuclei finds a sql injection vulnerability using the sqli-error-based template, and correctly identifies the database as being sqlite.

In the browser this request looks like this



Lets investigate the template nuclei is running. It's actually very simple, compared to other templates.

The template is: error-based-sql-injection

<https://raw.githubusercontent.com/geeknik/the-nuclei-templates/main/error-based-sql-injection.yaml>

The template sends a single ‘ to our endpoint then triggers if a recognizable database error is returned. This is a common error based sqli test.

```
requests:  
  - method: GET  
    path:  
      - "{{BaseUrl}}/"  
_____
```

In our case, it was likely “SQLITE_ERROR”

```
- "Pdo[./_\\\\\\/]firebird"  
# SQLite  
- "SQLite/JDBCDriver"  
- "SQLite\\.Exception"  
- "(Microsoft|System)\\.Data\\.SQLite\\.SQLiteException"  
- "Warning.*?\\W(sqlite_|SQLite3::)"  
- "\\[SQLITE_ERROR\\]"  
- "SQLite error \\d+:"  
- "sqlite3.OperationalError:"  
- "SQLite3::SQLException"  
- "org\\.sqlite\\.JDBC"  
- "Pdo[./_\\\\\\/]Sqlite"  
- "SQLiteException"  
# CND MsvNnR
```

The real strength is nuclei’s ability to runs 100s of templates like this one against 100s of known endpoints, incredibly fast. We’ve only tested one!

Read More:

- <https://github.com/projectdiscovery/nuclei>
- <https://corgea.com/Learn/all-you-need-to-know-about-dast-in-2025-comprehensive-guide>
- <https://projectdiscovery.io/blog/nuclei-fuzzing-for-unknown-vulnerabilities>