MAY 26, 2018 - 4 MINUTE READ - COMMENTS - ENUMERATION

On Username Enumeration

Enumeration is *identifying valid resources in a target address space*. This sounds academic and I hate academic mumbo jumbo but bear with me.

Username enumeration is *identifying valid user identifiers in an application*. These are typically usernames or account IDs.

Enumeration is an interesting topic. The concept of enumeration is straightforward but not always obvious. In this post I will explain username enumeration using an example.

Login Requests and Username Enumeration

Who am I?

I am Parsia, a security engineer at Electronic Arts.

I write about application security, reverse engineering, Go, cryptography, and (obviously) videogames.

Click on About Me! to know more.

Collections

One of the easiest places to check for username enumeration (and one of the first things I check) is the login request.

Send two unsuccessful login requests, one with a valid username (and invalid password because we do not want to login) and the other with an invalid one. **Compare the responses and see if you can spot a difference**. It there is a difference then you got username enumeration.

Sometimes it's part of the business logic. For example, Google or Amazon don't care about username enumeration. You enter your username and if valid, you are redirected to password page.

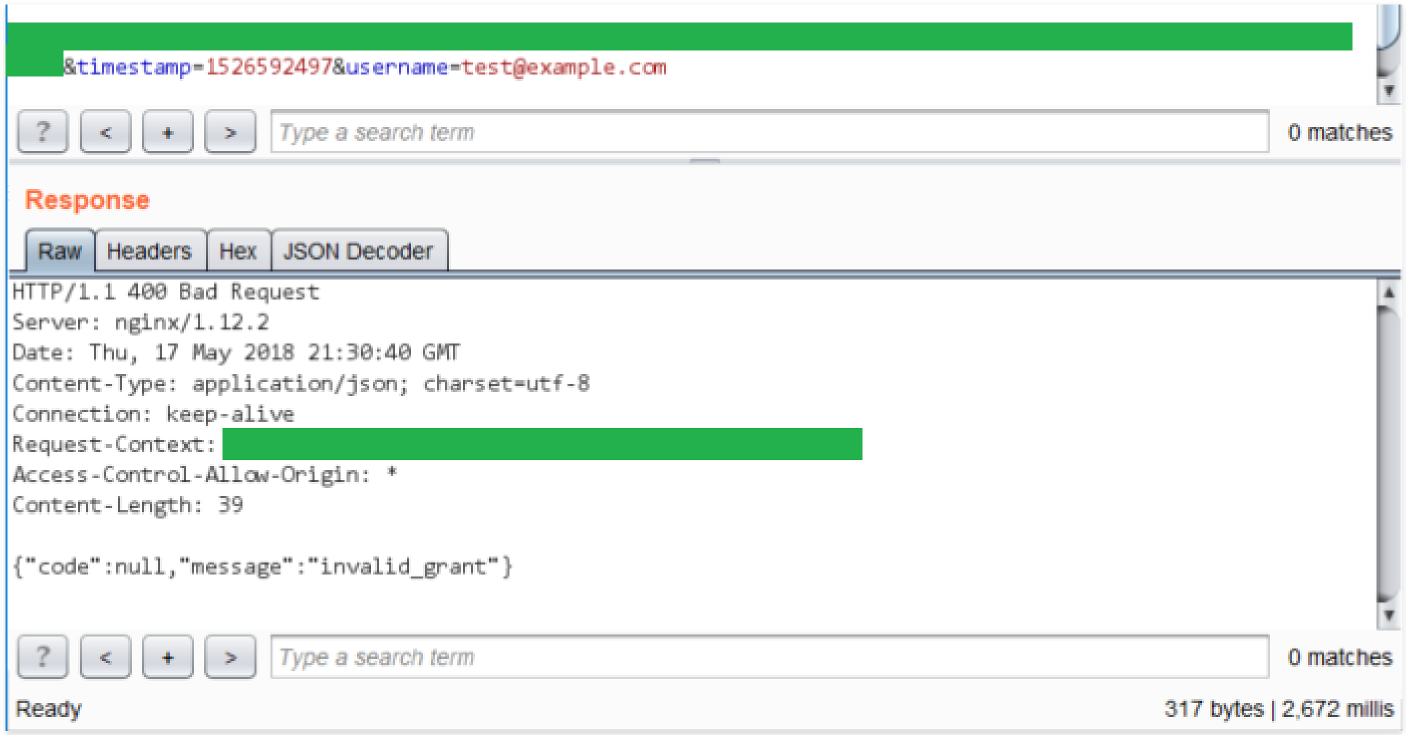
Sometimes the difference in the response message is pretty obvious. Something like "this username does not exist." Often times it's a bit more subtle and there's some numeric error code that gives it away.

The Curious Example

In this application, usernames are emails. This is pretty typical in most web application.

First I sent a login request with an invalid username. Here's the response displayed in Burp's Repeater tab.

Thick Client Proxying
Go/Golang
Blockchain/Distributed Ledgers
Automation
Reverse Engineering
Crypto(graphy)
CTFs/Writeups
WinAppDbg
AWSome.pw - S3 bucket squatting - my very legit branded vulnerability

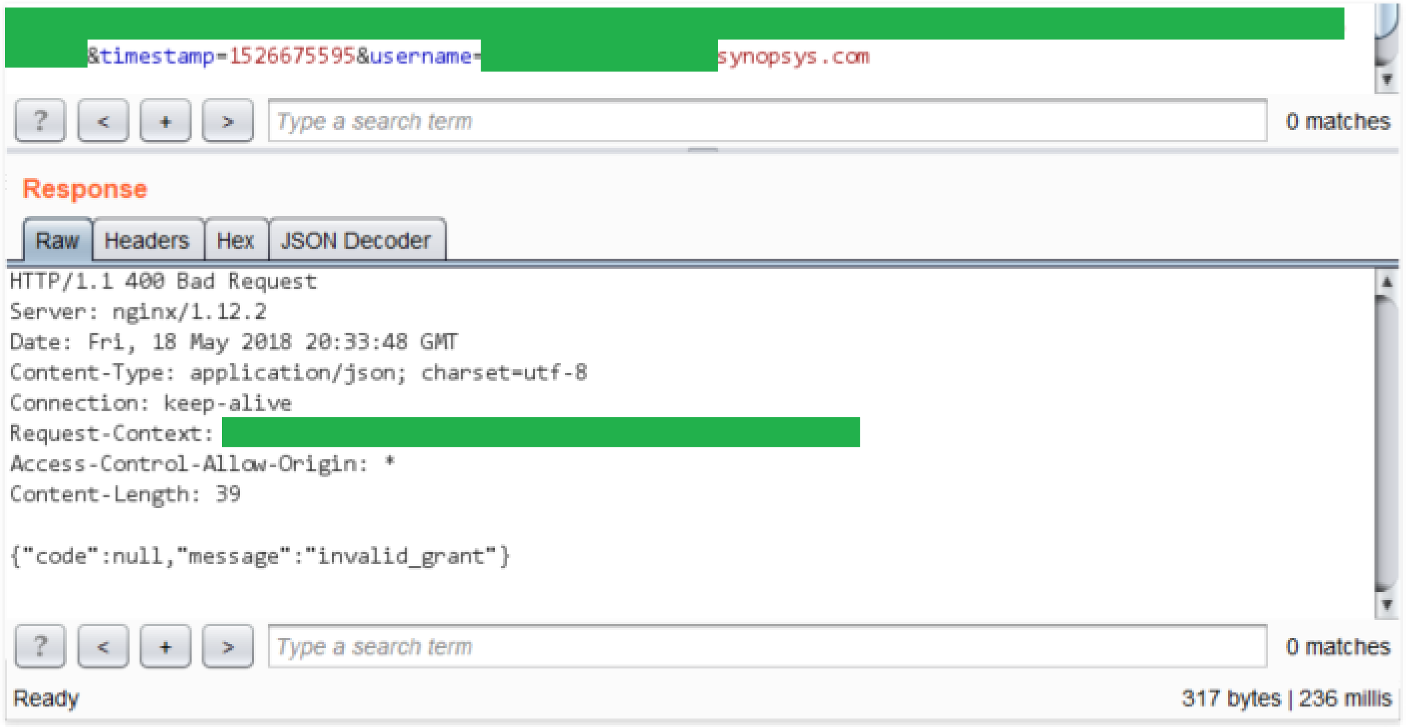


```
[redacted]&timestamp=1526592497&username=test@example.com
```

```
HTTP/1.1 400 Bad Request
Server: nginx/1.12.2
Date: Thu, 17 May 2018 21:30:40 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Request-Context: [redacted]
Access-Control-Allow-Origin: *
Content-Length: 39

{"code":null,"message":"invalid_grant"}
```

Response to login with invalid username

Then sent a request with a valid username and got a response.



[redacted]

```
[redacted]&timestamp=1526675595&username=[redacted]synopsys.com
```

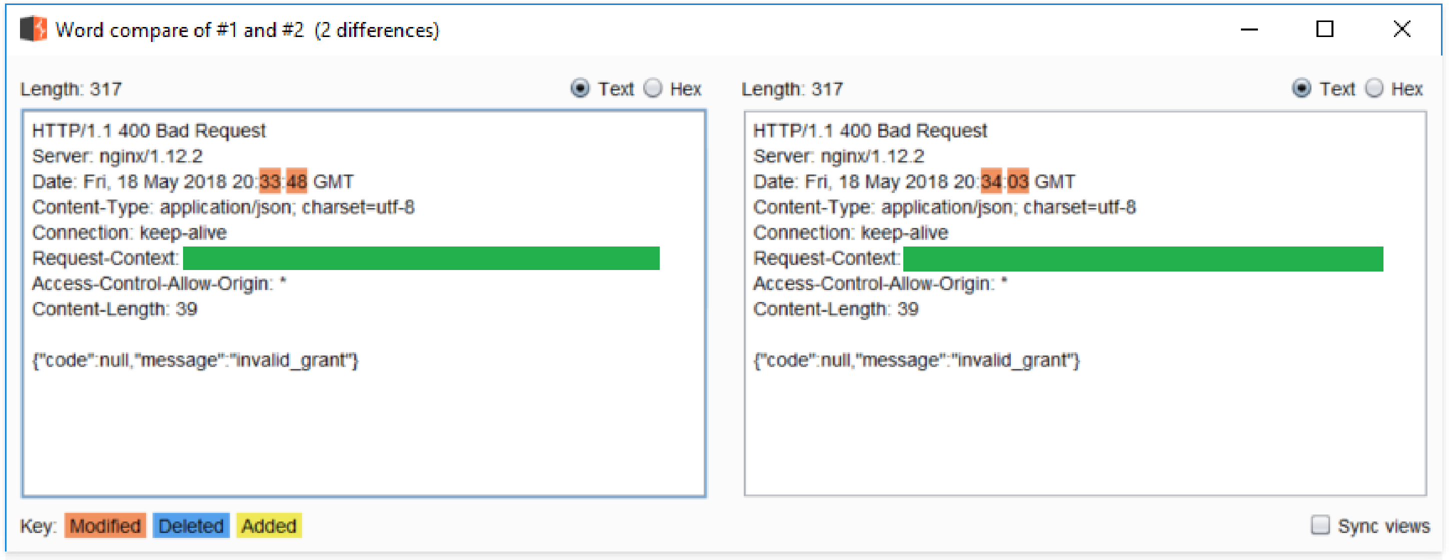
? < + > Type a search term 0 matches
Response
Raw Headers Hex JSON Decoder

```
HTTP/1.1 400 Bad Request
Server: nginx/1.12.2
Date: Fri, 18 May 2018 20:33:48 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Request-Context: [redacted]
Access-Control-Allow-Origin: *
Content-Length: 39

{"code":null,"message":"invalid_grant"}
```

? < + > Type a search term 0 matches
Ready
317 bytes | 236 millis

Response to login with valid username

This two look very similar. To easily spot differences, I used Burp's `Comparer`. Right click on each, `Send to Comparer` and we can compare `by words` there.


Word compare of #1 and #2 (2 differences)
Length: 317
[✓ Text] Hex
Length: 317
[✓ Text] Hex
Key: Modified Deleted Added
Sync views

```
HTTP/1.1 400 Bad Request
Server: nginx/1.12.2
Date: Fri, 18 May 2018 20:33:48 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Request-Context: [redacted]
Access-Control-Allow-Origin: *
Content-Length: 39

{"code":null,"message":"invalid_grant"}
```

```
HTTP/1.1 400 Bad Request
Server: nginx/1.12.2
Date: Fri, 18 May 2018 20:34:03 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Request-Context: [redacted]
Access-Control-Allow-Origin: *
Content-Length: 39

{"code":null,"message":"invalid_grant"}
```

Comparing two responses

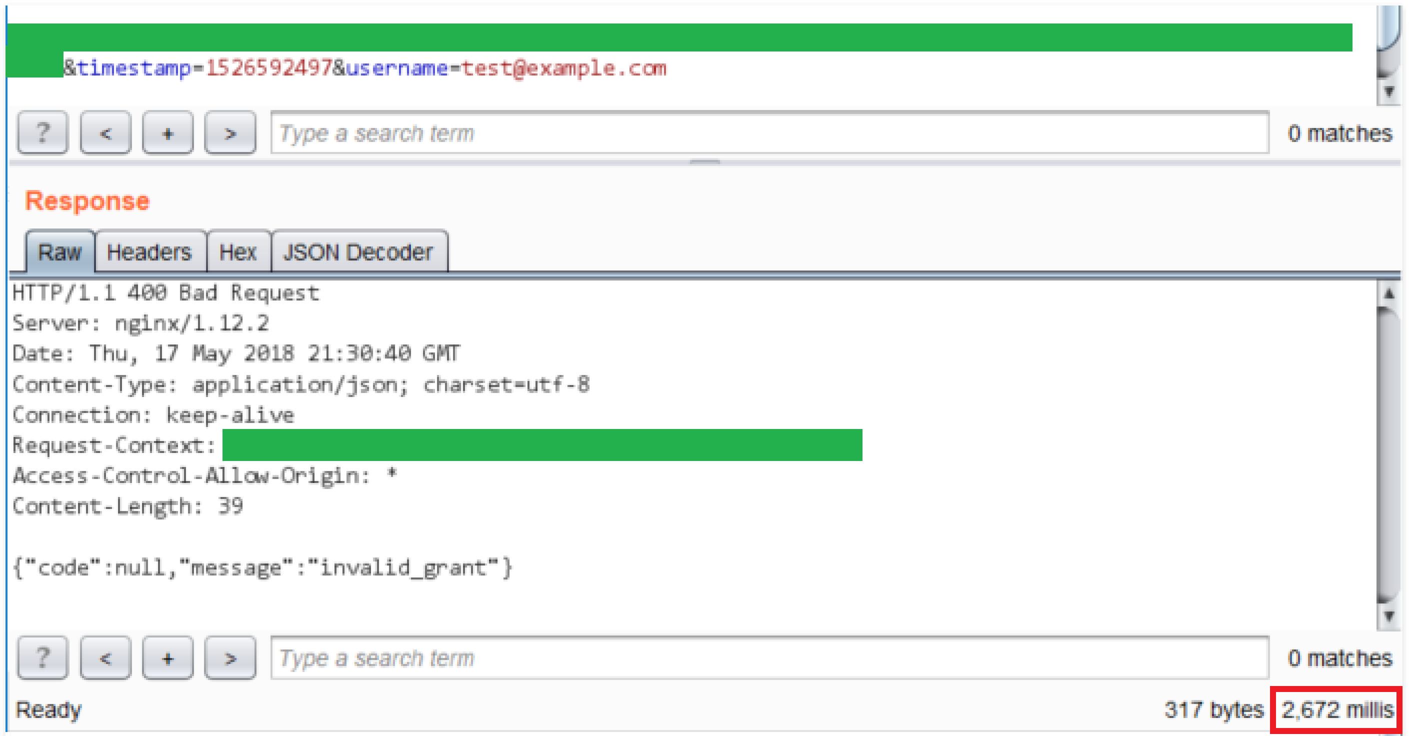
Well they are pretty much the same. The `Request-Context` is redacted from the image but the value is the same for both responses.

There is username enumeration there, it's not just as visible as it is. But don't worry, this is not some complex math. You will be able to spot it after it's pointed out forever.

Side-Channel Attacks

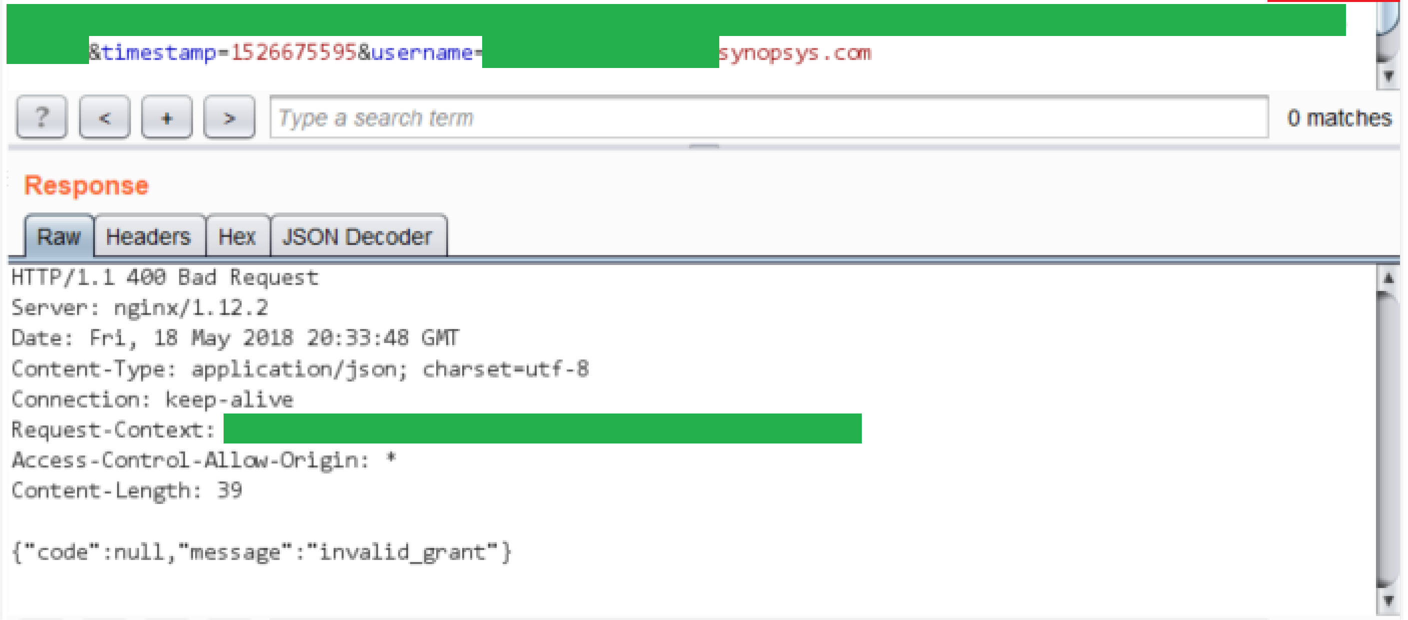
Side-channels are, well side-channels. These are channels from which the system can leak information. The most popular side-channel is time. This results in timing attacks.

Apart from the request/response, Burp shows us the response time. Unfortunately, this does not show up in `Comparer`. Let's put those initial responses together and look at the response times again.



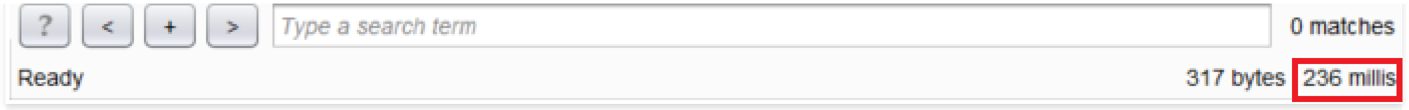
```
[redacted]&timestamp=1526592497&username=test@example.com
```

```
HTTP/1.1 400 Bad Request
Server: nginx/1.12.2
Date: Thu, 17 May 2018 21:30:40 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Request-Context: [redacted]
Access-Control-Allow-Origin: *
Content-Length: 39

{"code":null,"message":"invalid_grant"}
```



```
[redacted]&timestamp=1526675595&username=[redacted]synopsys.com
```

```
HTTP/1.1 400 Bad Request
Server: nginx/1.12.2
Date: Fri, 18 May 2018 20:33:48 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Request-Context: [redacted]
Access-Control-Allow-Origin: *
Content-Length: 39

{"code":null,"message":"invalid_grant"}
```



Response difference highlighted

Login with valid username takes around 250ms while invalid usernames take 10x more around 2.5s. This can be effectively used to weed out valid/invalid usernames/emails.

Appendix: Some Tips and Tricks

I have seen some timing attacks. I have read about others. This section shares what I have learned.

Reducing the Address Space

Know your address space. Research the address space of the what you are trying to attack. If the application uses emails, random strings do not work. Emails have specific formats. Same can be said about other structured data. Credit cards are 16 digits and the first six digits are predictable based on the issuer. Last digit is the `check digit` based on the Lunh Algorithm.

Like fuzzing, you don't want to pass invalid data. You want to bruteforce intelligently.

Bruteforcing

Side-channel attacks and bruteforcing are very closely connected. Usually a side-channel attack allows you to bruteforce something. In this case, we are bruteforcing email addresses. Reducing the address space allows us to bruteforce much better.

Let's assume we want to bruteforce a number. If our number is an `int`, we have a much easier time compared to `float`s. Theoretically the number of *Natural Numbers* is infinity and the number of floats between two ints is the same. In action, you can only implement so many decimal and floating points. The total number of ints is lower than total number of floats.

Conclusion

LOOK AT THE RESPONSE TIME.

Posted by Parsia • May 26, 2018 • Tags: tags2

Learning Go-Fuzz 2: goexif2 ContextIS xmas CTF Writeup

0 Comments Parsiya 1 Login

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS