SECURITY ASSESSMENT REPORT

Customer:
WideOpenWeb

Presented by :
PhishNet Security

Table of Contents

1. EXECUTIVE SUMMARY

During the period between 2024-12-11 and 2024-12-15, PhishNet Security conducted a security assessment of **WideOpenWeb**.

The purpose of this assessment was to evaluate the current security status and uncover vulnerabilities in the WideOpenWeb Web Application.

WideOpenWeb provides software, training, and research, to those dedicated to keeping their organization safe and are a global leader in cybersecurity. WideOpenWeb created Usurp Suite, the leading toolkit for web application security testing and handles large databases with a substantial amount of web traffic, therefore the security assessment aimed to identify potential attacks that attackers could exploit to leak user data or compromise the web application's integrity. This report presents the findings of the assessment, providing technical details about the identified vulnerabilities along with recommendations for their mitigation.

1.1. Results

The security assessment uncovered multiple vulnerabilities that could potentially be exploited to manipulate functionalities, compromise the web application, or gain unauthorized access to user information. These vulnerabilities included mechanisms allowing attackers to deploy malicious code, violating the server integrity or targeting users to compromise their sessions.

All the vulnerabilities found, were caused by the same factor. With the primary issue being the application's lack of input validation, allowing insertion of invasive code in various locations.

1.2. Recommendations

Strengthen the application's overall security framework by incorporating preventive measures against identified vulnerabilities.

Improve server and application configuration to meet security best practices.

Validate user input, escaping characters and commands that are not a part of the application's structure.

Conduct subsequent testing with automated methods included to potentially reveal further vulnerabilities.

Encrypt stored user information to minimize damages in case of intrusion.

2. FINDINGS AND RECOMMENDATIONS

This section will list the findings from the security assessment performed by PhishNet Security. Included are universal security recommendations for each vulnerability in the interest of improving WideOpenWeb's overall security framework. More detailed descriptions and recommendations for each corresponding vulnerability will be presented in section 3.

2.1. Approach to testing

PhishNet Security was given the task to perform a **web application assessment**. Conducting web application assessments is a critical step in protecting the security and reliability of online platforms. These evaluations involve an in-depth analysis of a web application's code, architecture, and configuration to uncover potential weaknesses or vulnerabilities. The goal is to detect and address security flaws proactively, preventing exploitation by attackers. This process encompasses various methodologies, including vulnerability scans, penetration tests, and detailed web application security evaluations. Such assessments often highlight vulnerabilities, including those categorized in the OWASP Top 10 Web Application Security Risks[1]. These top ten risks represent a broad consensus about the most critical security risks to web applications and serve as a guideline in the most crucial vulnerabilities to look for during testing.

Due to resources, only one assessor conducted the testing. This was a free assessment done by an apprentice whose knowledge was limited to manual testing. Therefore, it is recommended to perform subsequent tests with automatic testing included to reveal further potential vulnerabilities.

The assessment was conducted using solely Burp Suite Community Edition.

[1] https://owasp.org/www-project-top-ten/

2.2. Findings and Recommendations

The assessment revealed four different types of injection attacks: **SQL Injection, OS Command Injection, DOM based XSS** (Cross-Site Scripting) and **XML Eternal Entity Injection** (XXE). While these injections are unique in their own way and executed differently, the root cause all stem from targeting the same weakness. All four vulnerabilities found and listed in this report, were caused by the application's lack of user input validation.

Without robust user input validation, WideOpenWeb is vulnerable to attacks such as: data breach, a compromise of the application and hijacking of user sessions.

To prevent risks associated with **SQL injections**, the proactive method is the utilization of prepared statements with parameterized queries that separate SQL code from user input, ensuring inputs are treated as data instead of executable code.

DOM-based XSS are mitigated by avoiding directly inserting untrusted user input into the DOM. Re-evaluate the usage of insecure practices like innerHTML and use alternative safer methods where applicable.

For **XML Eternal Entity Injection** attacks, key measures include disabling external entity processing in XML parsers by configuring them securely. Restrict application access to sensitive files to limit the impact of potential attacks.

To mitigate **OS Command Injections**, avoid executing system commands directly with user input. If OS commands must be executed, validate all inputs using strict whitelisting and escape potentially dangerous characters.

By addressing these vulnerabilities WideOpenWeb can substantially improve their security framework.

2.3. Delimitations and Restrictions

Although WideOpenWeb granted PhishNet Security express written consent to conduct testing, neither open-source code nor test accounts were provided.

3. RESULTS AND RECOMMENDATIONS

3.1. Severity ratings

Severity	Description
High	Security vulnerabilities that can give an attacker total or partial control over a system or allow access to or manipulation of sensitive data
Medium	Security vulnerabilities that can give an attacker access to sensitive data but require special circumstances or social methods to fully succeed.
Low	Security vulnerabilities that can have a negative impact on some aspects of the security or credibility of the system or increase the severity of other vulnerabilities, but which do not by themselves directly compromise the integrity of the system.
Info.	Informational findings are observations that were made during the assessment that could have an impact on some aspects of security but in themselves do not classify as security vulnerabilities.

Table 1: Severity ratings.

3.2. Outline of identified vulnerabilities

Vulnerability	High	Medium	Low	Info.
SQL Injection	✓			
OS Command Injection	✓			
DOM based XSS		✓		
XXE Injection	✓			

Table 2: Identified vulnerabilities.

3.3. Technical descriptions of findings

3.3.1. SQL Injection

Severity: High

Description

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. This can allow an attacker to view data that they are not normally able to retrieve. This might include data that belongs to other users, or any other data that the application can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

The application contains a significant SQL Injection vulnerability that could allow an attacker to manipulate the database, potentially leading to data theft, data corruption, or unauthorized access to sensitive information.

During the security assessment it was discovered that querying the categories on the WideOpenWeb shop, is susceptible to SQL injections.

In the Example below, a GET request for the categories can be seen that retrieves the table names from the SQL database with the added payload **'UNION+SELECT+table_name,+NULL+FROM+information_schema.tables--**

```
GET /filter?category='UNION+SELECT+table_name,+NULL+FROM+information_schema.tables-- HTTP/2

Host: 0a9600d504695f328087d540002f00f5.web-security-academy.net

Cookie: session=[...]
```

Example 1: GET request with the initial SQL injection.

This listed all the table names, one of interest being:

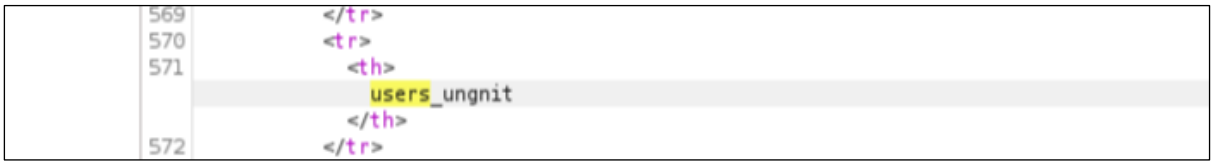

Image 1: Result of the initial SQL injection.

With the name of a table possibly containing user related info, further queries were made regarding the table name. The next payload:

'UNION+SELECT+column_name,+NULL+FROM+information_schema.columns+WHERE+table_name+=+'users_ungnit'-- revealed the columns where the table name **users_ungnit** is present.

```
GET
/filter?category='UNION+SELECT+column_name,+NULL+FROM+information_schema.columns+WHERE+table_name+=+'us
ers_ungnit'-- HTTP/2

Host: 0a9600d504695f328087d540002f00f5.web-security-academy.net

Cookie: session=[...]
```

Example 2: GET request with the second SQL injection.

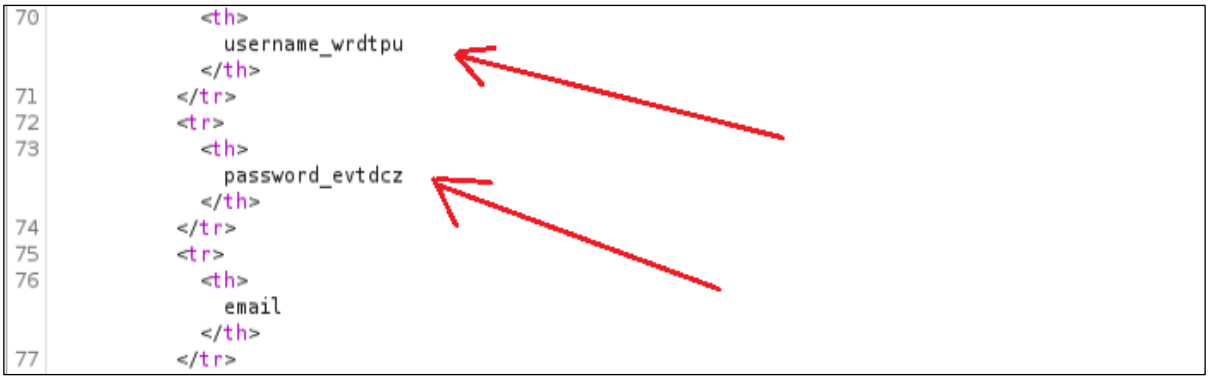

Image 2: Result of the second SQL injection.

The final payload **'UNION+SELECT+password_evtdcz,+username_wrdtpu+FROM+users_ungnit--** revealed the contents of the columns **password_evtdcz** and **username_wrdtpu** stored passwords and usernames, including the administrator account and password.

```
GET /filter?category='UNION+SELECT+password_evtdcz,+username_wrdtpu+FROM+users_ungnit-- HTTP/2

Host: 0a9600d504695f328087d540002f00f5.web-security-academy.net

Cookie: session=[...]
```

Example 3: Third and final SQL injection.

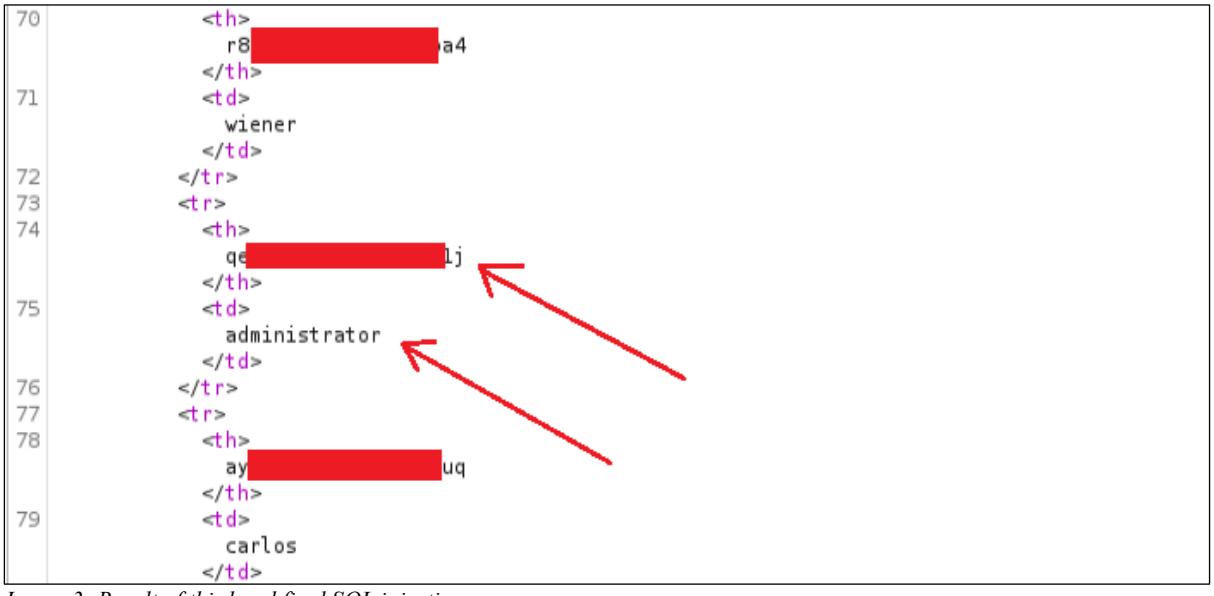

Image 3: Result of third and final SQL injection.

Image 4: Logged in to the administrator account with the retrieved details.

Recommendations

Implement prepared statements with parameterized queries, which separate the user input from the SQL logic and prevent any malicious input from being interpreted as part of the query.

Although prepared statements are by far the best preventative measure against SQL injections, to minimize further risks it is recommended to encrypt stored user information such as hashing passwords. This adds another layer of protection and is a good security practice overall.

For more information on SQL Injection prevention, see:

OWASP SQL Injection Prevention Cheat Sheet[2].

[2] https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

3.3.2. OS Command Injection

Severity: High

Description

OS command injection is also known as shell injection. It allows an attacker to execute operating system (OS) commands on the server that is running an application, and typically fully compromise the application and its data. Often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, and exploit trust relationships to pivot the attack to other systems within the organization.

WideOpenWeb's shop was found vulnerable to an OS command injection. The submit feedback function did not validate user input, specifically the email parameter in the POST request, causing the application to be susceptible to an injection. It was noted that the images on the website are stored in a public folder. Usually these folders default to /var/www/images/. In the example below, a POST request for submitting feedback is shown with the email parameter being manipulated with the following payload: **||whoami>/var/www/images/output.txt||**

```
POST /feedback/submit HTTP/2

Host: 0a08006b03f9445181dcb19a00db0093.web-security-academy.net

Cookie: session=[...]

csrf=C1[...]g7&name=sadas&email=||whoami>/var/www/images/output.txt||
```

Example 4: OS Command Injection in the email parameter

The payload will ignore the email submission, try to run the command **whoami**, which displays the current domain and username (as a proof of concept in this case). The command will get redirected to a file(output.txt) stored in /var/www/images/.

This gave a HTTP response code 200, confirming that the folder where images are stored was /var/www/images/. To verify that the injection was successful, an image source from the website value was altered (in this example filename=75.jpg) to **output.txt** resulting in the response of the **whoami** command being shown.

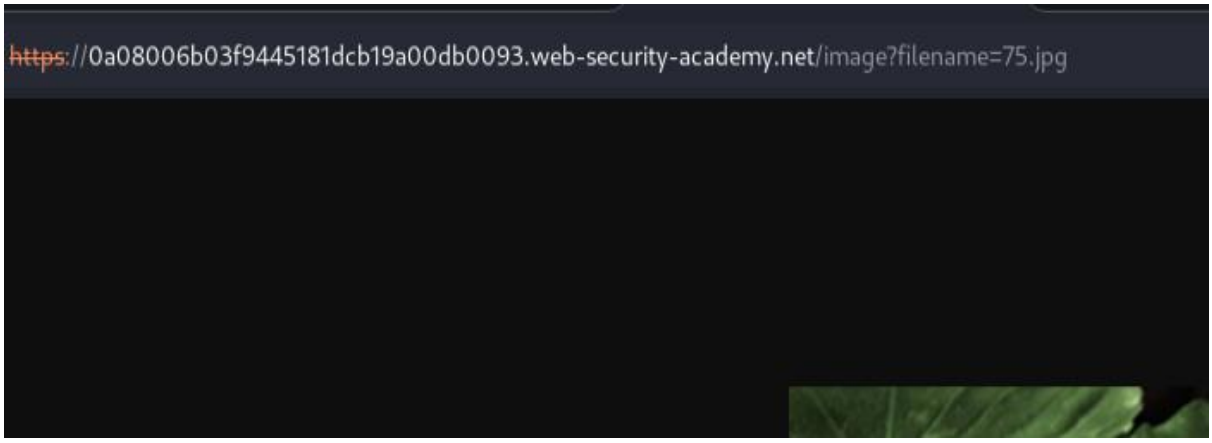

Image 5: Locally stored image from the WideOpenWeb shop opened in browser.

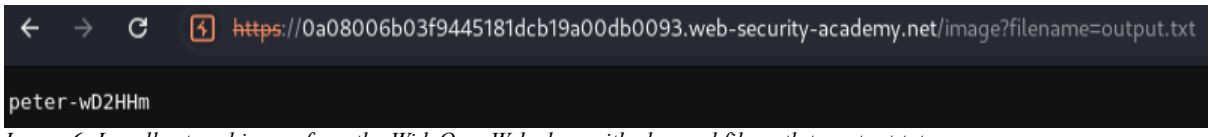

Image 6: Locally stored image from the WideOpenWeb shop with changed file path to output.txt.

Recommendations

In most cases, disabling OS Commands in user input is recommended.

If OS Commands are vital, implement strict whitelisting of characters and commands. Only allowing input that will not compromise the application.

For more information on OS Command Injection, see:

OWASP OS Command Injection Defense Cheat Sheet[3].

[3] https://cheatsheetseries.owasp.org/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.html

3.3.3. XXE Injection via Image File Upload

Severity: High

Description

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. In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks.

In WideOpenWeb's blog posts, it was possible to leave a comment with the option of uploading an image as an avatar. It was noted that the image upload supports **.svg** formats. Scalable Vector Graphics (SVG) is an XML-based vector image format for defining two-dimensional graphics, having support for interactivity and animation. Being XML based could potentially mean that it is also susceptible to XXE injections. A .svg file was created and uploaded with the following code:

```
<?xml version="1.0" standalone="yes"?>

<!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/hostname" > ]>

<svg width="128px" height="128px" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">

<text font-size="16" x="0" y="16">&xxe;</text></svg>
```

Example 5: Created .svg file containing malicious code.

This code printed the content of /etc/hostname to the avatar, exposing the name of the underlying server serving the website as a proof of concept that the injection was successful.

Image 7: Uploading the malicious .svg file as an avatar.

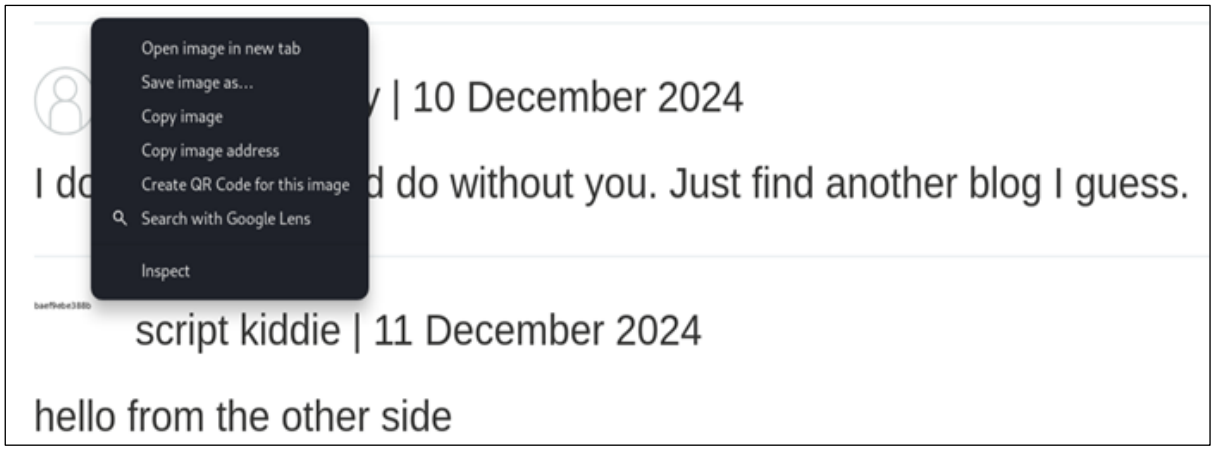

Image 8: Submitted comment showing an avatar.

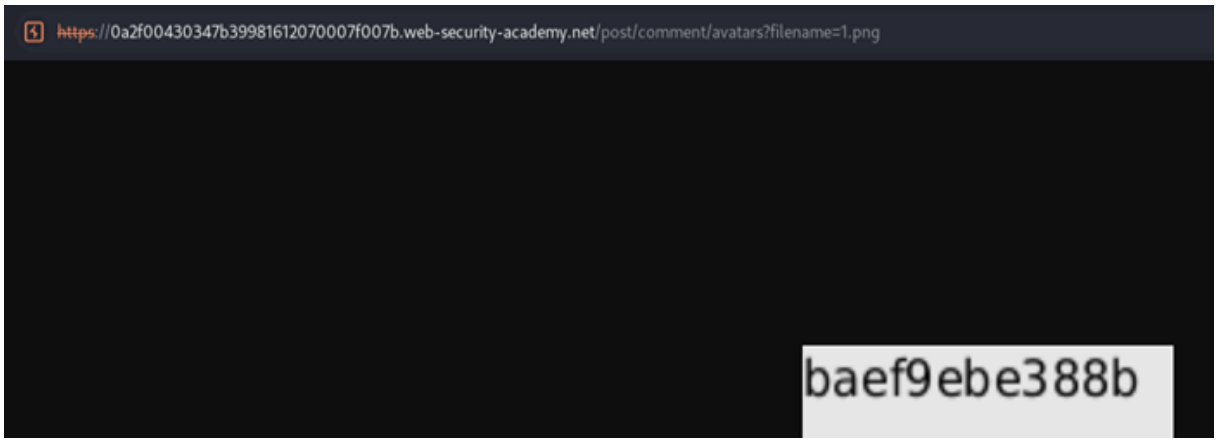

Image 9: The avatar image opened, revealing the /etc/hostname/.

Recommendations

Use whitelisting approaches to only allow known safe XML structures, elements, and attributes.

Utilize secure XML parsing libraries that do not automatically resolve external entities by default.

Restrict access to sensitive resources and ensure that XML parsing operations are performed with the least privileges necessary.

For more information on XML External Entity Prevention, see:

OWASP XML External Entity Prevention Cheat Sheet[4]

[4] https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

3.3.4. DOM-based XSS (Cross-Site Scripting)

Severity: Medium

Description

DOM-based XSS (Cross-site scripting) vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts.

The WideOpenWeb shop was found vulnerable to a DOM-based XSS due to user input not being handled correctly. This specific vulnerability was found by manipulating the functionalities of the window.postMessage() method. The window.postMessage() method enables cross-origin communication between Window objects, e.g., between a page and a pop-up that it spawned, or between a page and an iframe embedded within it. With this method and exploitation of the application's code structure, it was possible to execute a DOM-based XSS on the WideOpenWeb shop. The innerHTML in the code shown below is susceptible to an attack, allowing malicious insertion of HTML and JavaScript.

```
54      <!-- Ads to be inserted here -->
55      <div id='ads'>
56      </div>
57      <script>
58        window.addEventListener('message', function(e) {
59          document.getElementById('ads').innerHTML = e.data;
60        }
        )
61        </script>
```

Image 10: Vulnerable code

From another server, it was possible to send a web message to WideOpenWeb's website, said message will receive the value 'ads'. Due to the input not being validated, it was possible to insert code exploiting the lack of HTML-escaping and origin-checking. In the example shown below, a demonstration of how foreign code hosted on an exploit server can be used to execute a DOM-based XSS on the WideOpenWeb shop.

```
<iframe src="https://0a18003f04d3b24b81538ef80054007b.web-security-academy.net/" on-
load="this.contentWindow.postMessage('<img src=1 onerror=print()>','*')">
```

Example 6: Payload hosted on the exploit server.

This resulted in the iframe loading, the postMessage command sending a web message to the home page, the EventListener receiving the content of the message, and inserting it to the websites HTML. Due to the payload containing an image source which does not exist, it threw an error which executed the error handler of the payload, the print() command. The asterisk(*) allowed for the postMessage command to be dispatched from any origin, since the EventListener did **not** have any origin validation.

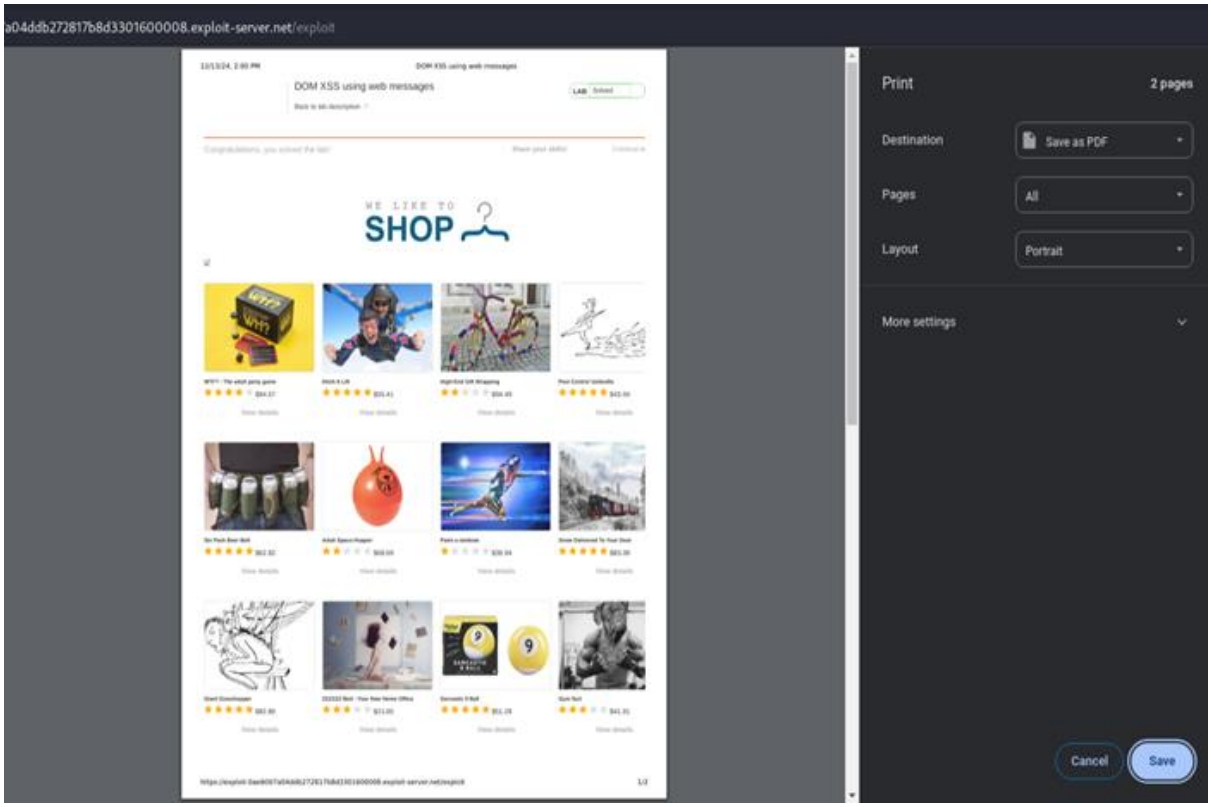

Image 11: WideOpenWeb shop after the message was sent containing the payload.

Recommendations

Untrusted data should be encoded and/or escaped according to the context in which it is included, to prevent it from being treated as part of the page code structure.

Validate the origin of the web message, allowing only messages from a trusted URLs.

For more information on DOM-based XSS prevention, see:

OWASP DOM based XSS Prevention Cheat Sheet[5]

[5] https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html