# Cyber Security Fundamentals (M)

Capture the flag assessment report

Umakhihe Arnold (2445734U)
Elumeziem Chika (2500799E)
Anne-Marie Gill (2431989G)
Binta Shuwa (2407958S)
Bokyung Lee (2431088L)

# Table of Contents

# Introduction

This report analyses various vulnerabilities, and their exploits, discovered in the OWASP Vulnerable Web Application Project. Firstly, the vulnerabilities are identified and analysed. Secondly, exploits are demonstrated. Thirdly, possible solutions to mitigate the threats posed by the vulnerabilities are proposed, along with an analysis of its effectiveness.

# Attack stage of OWASP Vulnerable Application project

## BodgeIT

Through completing the first 10 challenges, the following vulnerabilities were discovered on the BodgeIT website.

### *Insecure Password Creation*

In challenge 1, the team was able to access the account test@bodgeitstore.com by entering "password" in the password login field. BodgeIT has insufficient requirements for user passwords which leaves accounts vulnerable to brute force common password attacks.

### *SQL Injection*

The login form was found to be vulnerable to SQL injection, enabling the team to hack into user accounts. In challenge 2, to login as user1@thebodgeitstore.com, the username was entered into the username field and ' OR '1'='1 was entered as the password. This likely evaluates to: SELECT * FROM USERS WHERE USERNAME='user1@thebodgeitstore.com' AND PASSWORD='' OR '1'='1'. The password field will always evaluate to true, therefore the query on the backend becomes: SELECT * FROM USERS WHERE USERNAME='user1@thebodgeitstore.com'.

For challenge 3, to login as admin@thebodgeitstore.com, the SQL code ' OR '1'='1 was entered at the end of the username and the password field left blank. The first single quote would serve to close the first query string, then '1'='1' would always evaluate to true. The likely SQL query on the backend would be: SELECT * FROM USERS WHERE USERNAME='admin@thebodgeitstore.com' OR '1'='1' AND PASSWORD=''. This would evaluate to: SELECT * FROM USERS WHERE PASSWORD=''.

The query above will return a user with the specified password. If the password field is tested with various values, the user accounts with the same password as the specified password would be returned. For instance, password “password” returns test@thebodgeitstore.com, while “' OR '1'='1” returns user1@thebodgeitstore.com. When random and unlikely passwords are used, the admin user account is always returned. In this case, it can be argued that the program is designed to return the admin user when the password is not found.

### *Poor Authorization and Access Management*

In challenge 4, the team was able to access the admin page without authorization by entering the location of the admin page into the URL. Additionally, in challenge 5, the team was able

to view diagnostic data of the basket by adding "?debug=true" to the end of the URL on this page.

### *Cross Site Scripting (XSS)*

Many of the forms on the website were found to be vulnerable to cross site scripting attacks. In challenge 6, by entering <script> alert("XSS") </script> in the search bar, the team executed a reflected XSS attack to display an alert box.

The team created a user account <script> alert("XSS") </script>@bodgeitstore.com and were able to solve challenges 7 and 8, by accessing pages where this username was retrieved from the database to the client. These were stored XSS attacks.

In challenge 10, the XSS vulnerability on the search page was exploited to create a URL that automatically adds a product to a website visitor's cart. Script attached in appendix.

### *Insecure Cookies*

The team solved challenge 9, by changing the basket id in the cookies to the basket id of another user. The basket id for users who are not logged in is stored in a cookie called b_id. The value of b_id can be altered by a web user. If the b_id of another user is known, the cookie value can be changed to that user's basket id and items will be added to their basket.

## WackoPicko

### *Command Line Injection*

The team was able to execute commands on the server, after noticing that the password check functionality at http://192.168.56.101/WackoPicko/passcheck.php uses shell command "grep" during execution. The team was able to execute dangerous commands by piping the command. Such commands include removing files and folders recursively "rm -rf …". For example, using "|rm -rf * #" as the password, removed all files and folder in the directory.

### *Inappropriate Use of GET*

User ids are set in GET requests and can be changed easily on the view.php page http://192.168.56.101/WackoPicko/users/view.php?userid=1. The team was able to gain unauthorised access to other users account by inserting multiple user ids in the URL.

WackoPicko requires user to make a purchase before getting access to high-quality pictures. However, the team was able to access high quality images without making a purchase by changing the picid parameter in the URL: http://192.168.56.101/WackoPicko/pictures/high_quality.php?picid=15&key=highquality.

### *Logic on the Coupon system*

After logging in to WackoPicko, logic flaw can be found in the coupon system. can be multiple times reduce final cost zero.

# Defence Mechanisms

## Defending Against XSS Attacks

Cross site scripting (XSS) attacks occur when scripts are submitted to a dynamic web application allowing the attacker to perform malicious activities, such as stealing sensitive information or performing session hijacking [1]. The three main types of XSS attacks are persistent or stored attacks, reflected attacks, and DOM based attacks [2].

The first section of the website vulnerable to XSS attacks was the search bar. In an example of a non-persistent XSS attack, the script that the attacker enters into the search field is sent in an HTTP GET request and "reflected" in the response to be executed by the client. To help protect the search bar against these attacks, an XSS filter should be created and applied. XSS filters looks for certain characters and sequences. It filters them out either by stripping them from the input or applying escaping to them i.e. applying HTML encoding to prevent them from being executed. Although a sanitization list may seem comprehensive, there are many ways that attackers try to bypass these filters such as by using malformed or nested tags, or executing the malicious code using event handlers which might not have been included in the filter [3].

Another XSS vulnerable form on the website was the registration form where attackers could use XSS in the email when signing up. The email field validates that the address has the "@someprovider.tld" sequence, but it should also have an XSS filter. Additionally, an email field can have extra validation since the form of an email address is more restrictive than a potential search term. Preventing someone from registering with a malicious script in the email field would ensure code will not be executed when the profile is clicked or when it is loaded on the admin page. These are examples of persistent XSS as they are stored in the database waiting to be executed.

Two additional security measures that could be used to protect against XSS attacks are using appropriate response headers and implementing a content security policy. Response headers are attached to http responses and create a detailed context for the response [4] and can be used to specify what the return content type should be. If the response header is application/json instead of text/html, then there is no active content in the response and the script cannot be executed. Another response header that is employed to protect against XSS attacks is the X-XSS-Protection response filter however this currently has limited browser support and some browsers are planning to drop support for it [5].The content security policy can be implemented as a response header or in meta tags and operates by defining which sources are approved to be loaded by the browser. A downside of the content security policy is that it currently does not have universal browser support [6].

## Defending Against Cookie Manipulation

In the BodgeIT app, baskets for users without an account are associated with a basket id stored in the cookie. An attacker could exploit XSS vulnerabilities to execute a script on the user's browser to get the cookie. They would then have access to the site user's basket. They could add items, remove items, and make unauthorized purchases.

In addition to securing the site against XSS attacks, session information should never be stored openly and unprotected in cookies. BodgeIT should use cookie attributes to increase the security of their cookies. They should use the https protocol and set cookies to have the secure flag which would ensure that they cannot be transferred over the insecure http protocol. The HttpOnly attribute should also be set, as this would help prevent an attacker from accessing cookies through an XSS attack. It would do this by blocking client-side cookie access. However, it should be noted this flag is not completely attack proof [7].

## Defending Against Weak Passwords

Weak passwords are generally vulnerable to dictionary attacks, brute force attacks, shoulder surfing techniques, and even guessing. To mitigate against poor password hygiene, websites should urge users to practise to good password hygiene.

Users should set strong passwords. Previous study shows that length alone is not sufficient for a secure password. A strong password should have at least 12 characters, contains alphabets, numbers and symbols, with numbers preferably in the middle not as a prefix or suffix of the password. It should also be memorable to ensure good usability practice. [8].

Users should also avoid the use of a single password for multiple accounts, activate Multi-Factor Authentication (MFA), which adds another layer of security in case a password has been compromised, cultivate the habit of changing passwords within memorable safe intervals, and use password managers.

## Defending Against SQL Injection

An SQL injection attack on a data driven web app occurs when a non-validated input containing malicious SQL query is inserted by an attacker through legitimate query statements, which is then executed by the database management system [9]. According to OWASP, SQL injection is ranked the topmost web application vulnerability [10]. Sadegian and colleagues categorised 3 detection and prevention techniques to mitigate against SQL injection [9].

1. Best Code Practices: This include a set of policies and standards to improve code quality and robustness for developers. Such techniques for this purpose include manual defensive coding strategies, use of SQL DOM and parameterised query insertions.
2. SQL Injection Detection: There are various techniques employed to detect SQL injections, some of them are SQL injection testing using static and dynamic analysis popularly called SQL UnitGen, Mutation Based SQL Injection Vulnerability Checking (MUSIC), and vulnerability and attack injection.
3. SQL Injection Runtime Prevention: Techniques and tools for this purpose include SQL Rand, use of applications using positive tainting and aware evaluation (WASP), and SQL proxy based blocker.

## Defending Against URL Manipulation Attack

Universal Resource Locator attack also known as Hyper Text Transfer Protocol Manipulation is an attack on web application systems to gain unauthorised access to information by direct URL manipulation. This can be done either by trial and error method or directory traversal

method. Preventive measures against URL manipulation is implemented through effective web server configuration.

Web masters should employ chroot mechanism which prevents against the browsing of pages found below the root of the website, delete unnecessary and or useless directories, files and configuration options on the server, deactivate the display of files found in directories that are not index files. (Directory Browsing), ensure the server implements adequate protection against access to directories with sensitive information, ensure severs efficiently and accurately interprets dynamic page requests especially with backup files (.bak), block HTTP viewing of HTTPS accessible pages, and ensure the URL format is not in a parameterised string query structure. When a parameter needed to be sent from a client to the server, ensure it is accompanied by a valid token. In cases where sensitive parameters could not be abstracted from the URL, it must be cryptographically protected.

### Defending Against Poor Authorization and Access Management

The vulnerable websites used hiding, and client-side code to implement access control. Access control should only be enforced in trusted server-side code, where the attacker cannot modify the access control check or metadata. To prevent poor access control, developers should deny access to functionality by default, use access control lists and role-based authentication mechanisms, and avoid just hiding functions. [12]

### Defending Against Bad Logic

Bad logic in code can be avoided by carrying out intensive testing for key functionalities. Developers should adhere to industry best practices when producing code, which should be rigorously tested by independent testers. This will help ensure that every core functionality works, as intended.

### Defending Against Command Line Injection

Command injection attacks are possible largely due to insufficient input validation. [13] The attacks executed on the web apps can be avoiding by sanitizing user input. Characters which can be used to escape the shell are “&&”, “;”, “|”, and “…”. These dangerous characters should be banned or sanitized on input fields, where the value would be appended to a shell command.

## Conclusion/Discussion

Through systematic testing of point with potentially weak security, several security vulnerabilities were found on both the BodgeIT and WackoPicko web applications. These vulnerabilities which include, but not limited to, SQL injection, cross site scripting, URL manipulation, poor authorisation and access management, and insecure cookies, suggest that the website can be very easily attacked, even by amateur hackers. In this report, several defence mechanisms were suggested to help defend against those vulnerabilities, and with proper implementation would help to reinforce the security of these web apps by increasing their defence against common attacks.

# References

[1] Engin Kirda 2011. Cross Site Scripting Attacks. In Encyclopedia of Cryptography and Security, Henk C A van Tilborg and Sushil Jajodia (eds.). Springer US, Boston, MA, 275–277.

[2] Shashank Gupta and B. B. Gupta. 2017. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. Int. J. Syst. Assur. Eng. Manag. 8, 1 (2017), 512–530

[3] OWASP. XSS Filter Evasion Cheat Sheet. Retrieved from https://owasp.org/www-community/xss-filter-evasion-cheatsheet

[4] Mozilla Web Docs. 2020. Response header. Retrieved from https://developer.mozilla.org/en-US/docs/Glossary/Response_header

[5] OWASP. Cross Site Scripting Prevention. Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[6] Brian Jackson. 2019. Hardening Your HTTP Security Headers. Retrieved from https://www.keycdn.com/blog/http-security-headers

[7] David Johansson. 2017. Cookie Security:Myths and Misconceptions. Retrieved from https://owasp.org/www-chapter-london/assets/slides/OWASPLondon20171130_Cookie_Security_Myths_Misconceptions_David_Johansson.pdf

[8] Shay, R., Komanduri, S., Durity, A.L., Huh, P., Mazurek, M.L., Segreti, S.M., Ur, B., Bauer, L., Christin, N. and Cranor, L.F., 2016. Designing password policies for strength and usability. ACM Transactions on Information and System Security (TISSEC), 18(4), pp.1-34.

[9] A. Sadeghian, M. Zamani and A. A. Manaf, "A Taxonomy of SQL Injection Detection and Prevention Techniques," 2013 International Conference on Informatics and Creative Multimedia, Kuala Lumpur, 2013, pp. 53-56.

[10] (OWASP), “O.W.A.S.P. Top 10 Vulnerabilities.”; Available from: https://www.owasp.org/index.php/Top_10 2013.

[11] Sharma, Pratima, and Bharti Nagpal. "A STUDY ON URL MANIPULATION ATTACK METHODS AND THEIR COUNTERMEASURES." (2015).

[12] Hdiv Security. Broken Access Control. Retrieved from https://hdivsecurity.com/owasp-broken-access-control

[13] OWASP. Command Injection. Retrieved from https://owasp.org/www-community/attacks/Command_Injection

# Appendix

## Proof of Completed BodgeIT Challenges

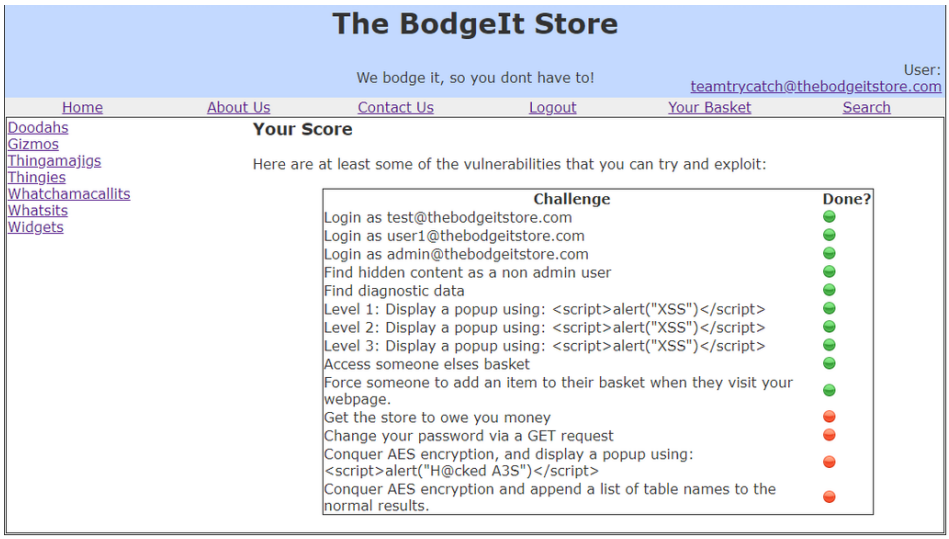


Figure 1: Figure of completed BodgeIT challenges

## Challenge 10 script and generated URL

*XSS Script*

<script>

window.addEventListener("DOMContentLoaded", (e)=>{

let body = document.getElementsByTagName("body");

body[0].innerHTML = "<form action='basket.jsp' method='post'><input type='hidden' name='productid' value='14'><input type='hidden' name='price' value='2.6'><input id='quantity' name='quantity' value='1' maxlength='2' size='2' style='text-align: right' readonly=''><input type='submit' id='submit' value='Add to Basket'></form>";

document.createElement('form').submit.call(document.getElementsByTagName("form")[0]);

});

</script>

*XSS URL*

http://192.168.56.101/bodgeit/search.jsp?q=%3Cscript%3E%20window.addEventListener(%22DOMContentLoaded%22,%20(e)=%3E%7b%20let%20body%20=%20document.getElementsByTagName(%22body%22);%20body%5b0%5d.innerHTML%20=%20%22%3Cform%20action=%27basket.jsp%27%20method=%27post%27%3E%3Cinput%20type=%27hidden%27%20name=%27productid%27%20value=%2714%27%3E%3Cinput%20type=%27hidden%27%20name=%27price%27%20value=%272.6%27%3E%3Cinput%20id=%27quantity%27%20name=%27quantity%27%20value=%271%27%20maxlength=%272%27%20size=%2

72%27%20style=%27text-
align:%20right%27%20readonly=%27%27%3E%3Cinput%20type=%27submit%27%20id=
%27submit%27%20value=%27Add%20to%20Basket%27%3E%3C/form%3E%22;%20docu
ment.createElement(%27form%27).submit.call(document.getElementsByTagName(%22form
%22)%5b0%5d);%20%7d);%20%3C/script%3E