Cyber Security Fundamentals Assessed Exercise

Team AA Final Report

Kevin Wu 0808148w

Claudia Ortiz-Duron 2412650o

Laimonas Samalius 2424831s

Sultan Altwijri 2312914a

Ali Rashed A Al Qarni 2286368a

Introduction

Over the course of this project a range of vulnerabilities in the BodgeIt web application were discovered. This report will provide a brief summary of each exploit and suggest potential protective mechanisms that the owner of BodgeIt might use to secure their web application against these attacks.

Exploit 1: Injection Flaws (Logging in as someone else)

The first vulnerability to be discovered was an injection flaw. This is a method that attackers use to take advantage of poorly coded web applications and can deliberately corrupt or delete data and cause a denial of service by locking tables [1]. In the case of BodgeIt, an attacker can also bypass authentication to log on as an authorised user, as the application can mistakenly process untrusted data from form fields, and execute a SQL database query [2]. For example, by typing test@thebodgeitstore.com' or ''=' into the log in box, the SQL query evaluates to true, and the attacker is able to login as 'test' if there exists an entry with that username in the 'user' table of the database. This method was also used to log in as Admin and User1.

A way to securing BodgeIt against this type of attack is by using parameterised queries and bounds. This method means that a query is executed using the values that are entered in the form fields, as opposed to being concatenated with SQL syntax [3]. In doing so, the likelihood of an attacker executing a valid SQL query on the database is reduced and therefore the integrity of the database is preserved. Furthermore, developers should not write dynamic queries. They should limit opportunities for users to input malicious SQL that could affect the logic of the executed query. In addition, the principle of the least privilege should be applied when providing accounts used to connect to the SQL database. If a web application only needs to retrieve web content from a database by means of SELECT statements, it is necessary to avoid connection credentials to other privileges such as INSERT, UPDATE, or DELETE privileges. [8]

Exploit 2: Broken Access Control

2.1 Find hidden content as a non-admin user

It was noted that the URL links all shared a similar pattern. Through guesswork, an attacker can simply access Admin page by directly targeting the Admin page URL. As the admin page did not require user authentication to access the contents of this page, an attacker would be able to view the web app owner's sensitive data. This type of vulnerability (Missing Function Level Access Control/Broken Access Control/Sensitive Data Exposure) could be prevented by denying access to all potential users initially, before granting specific access for each function associated with a role via authentication [14]. Taking this approach will ensure that all pages in the web application are checked and verified for appropriate required restriction level. Failed log in attempts could also be monitored to ensure where attacks are directed.

2.2 Getting the store to owe you money

This flaw was detected by exploring page code and amending it to add desirable value bypassing access control checks. An example of this attack is accessing the code directly

through the browser to modify it, locating a specific value and changing it. Protection against this attack, can be provided by denying default public resources. In addition, model access controls should require record ownership, preventing the user from creating, updating, or deleting any record.

Exploit 3: Security Misconfiguration (Find diagnostic data)

This attack was carried out simply by adding debug=true at the end of the URL. In this type of attack, in order to get unauthorized access or knowledge of the system, hackers exploit unpatched flaws. Possible attacks involve sample applications with security flaws that are not removed from the production server. The attacker could log in with default passwords, if one of these applications is in the admin console without changes in the default accounts. The revealed data [10] can provide information such as source code, hidden filenames, and detailed error messages.

This information can later reveal vulnerabilities and be used to help stage attack against the web application. This type of security misconfiguration risk could be mitigated by implementing a minimal platform without unnecessary features, components, documentation, samples and removing unused features and frameworks. [9] Additionally, prevention could increase [11] by ensuring that debug code is not left in production. The code should ensure that debug messages are not produced in the application. Instead of 'debug=true' URL parameter the global debug mode should be enabled.

Exploit 4: Cross-Site Scripting (displaying alert boxes via Javascript)

Another vulnerability that was discovered is called "Cross-Site Scripting" (XSS). In this example, it was possible to inject benign scripts into the web application, causing a pop-up alert after a client clicks on the submit button that reloaded the page. This method could be used by attackers to steal user accounts, read cookies and transmit unauthenticated data. The script communicates with the server, generates a response with the and leads the user's browser to run these scripts without any security restrictions. If the script is stored it should be permanently on the web server [4].

A way of securing BodgeIt against this type of attack is using software and libraries, such as Microsoft's Anti-Cross Site Scripting Library or OWASP's Java Encoder project that work by identifying and removing XSS vulnerabilities in a program and santising input by parsing the HTML page and making it safe to display. [15] For example, dangerous keywords such as the <script>, could be filtered out ensuring harmful scrips are not executed.

In addition, further protection could be provided by frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS. These libraries detect the limitations of XSS protection, and appropriately handle the use cases that are not covered. By enabling Content Security Policy (CSP) as defence-in-depth mitigation control against XSS, Cross-Site Scripting could also be avoided. [13]

Exploit 5: Broken Authentication/Session Management (Access someone else's basket)

The attack was carried out by altering the value of a basket's cookie. Due to a fault in session management functions, it was possible to change the basket's ID in the cookie and therefore access another user's basket. Protection against this type of attack, could be provided by limiting the basket ID storage in the session cookies. In addition, it is also

recommended that the session interval is set lower to lessen the time that an attacker has to use a valid session ID before it's invalidated on the server side [16]. Furthermore, all account management functions should require authentication even if the user has a valid session ID.

Exploit 6: Information exposure through query strings in url

Another vulnerability that was explored was the possibility for a user to change their password via a GET request, rather than using default POST method. The GET method is less secure as the data in the form fields is encoded into the URL [12]. This means that the password is logged in a variety of locations, such as the user's browser and the web server. Consequently, the password can be viewed by going through the user's browser history or using the back button [6].

The POST method is recommended where the form submission results in a page reload. The new password will not be encoded into the URL and is therefore more secure [7]. To implement this safety protocol, the web application owner should ensure that the appropriate tag - <form method="post" action="."><, is included in all html files containing a form.

Exploit 7: Cross Site Request Forgery

This vulnerability is known as a Cross-Site Request Forgery attack. It was flagged by being able to force someone to add an item to their basket via another webpage. An attacker can use this method to trick an unsuspecting victim who is logged on to a web app, to send a harmful request to the server by clicking a link, for example. The exploit makes use of the victim's session when the user is not required to be re-authenticated This attack is more dangerous when it is used against a user who has extra privileges, such as a superuser/admin.[18]

One way to lessen this risk, is by using a token based CSRF defence, either stateful or stateless. Each logged on user is given a token which is uniquely linked to their current session. These are sent in the html form any time a user clicks on a link and are sent to the server. The server will check that the token is valid, and only then process the request. This will protect the web app as it'll block any requests where the token is not valid. A user interaction-based protection in conjunction with token-based mitigation, would enforce this defence method.[17][19]

Conclusion

The team carried a lot of research to complete this task. There were many vulnerabilities with the BodgeIt applications. These have been discussed, and suggestions to secure the web app against them have been proposed. Working on this challenge presented to the group that hackers can inflict a high level of damage to any web application through many different methods. The owner of any web application should test if their web application is secure and ensure that the risk of attack is minimised. Taking steps to reduce vulnerabilities decrease risk of breaches, financial loss and loss of reputation. We believe that developers should be trained to develop more secure code that could provide better protection against these vulnerabilities.

References

[1] Owasp.org. (2019). [online] Available at: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf [Accessed 22 Feb. 2019].

[2] Veracode.com. (2019). [online] Available at: https://www.veracode.com/security/dot-net-sql-injection [Accessed 22 Feb. 2019].

[3] Microsoft.com (2019). [online] Available at: https://blogs.msdn.microsoft.com/sqlphp/2008/09/30/how-and-why-to-use-parameterized-queries [Accessed 22 Feb. 2019].

[4] Marashdih, A. and Zaaba, Z. (2017). Cross Site Scripting: Removing Approaches in Web Application. *Procedia Computer Science*, 124, pp.647-655. [Accessed 22 Feb. 2019].

[5] Owasp.org. (2019). [online] https://www.owasp.org/index.php/Information_exposure_through_query_strings_in_url [Accessed 22 Feb. 2019].

[6] Portswigger.net. [online] https://portswigger.net/kb/issues/00400300_password-submitted-using-get-method [Accessed 22 Feb. 2019].

[7] Jkorpela.fi. (2019). [online] http://jkorpela.fi/forms/methods.html [Accessed 22. Feb. 2019].

[8] [online] https://security.berkeley.edu/resources/best-practices-how-articles/system-application-security/how-protect-against-sql-injection[Accessed 22. Feb. 2019].

[9] [online] https://www.owasp.org/index.php/Top_10-2017_A6-Security_Misconfiguration [Accessed 22. Feb. 2019].

[10] [online] https://www.indusface.com/blog/owasp-top-10-vulnerabilities-2013/

[Accessed 22. Feb. 2019].

[11] [online]https://www.owasp.org/index.php/Error_Handling,_Auditing_and_Logging

[Accessed 22. Feb. 2019].

[12] Owasp.org. (2019) [online] Available at: https://www.owasp.org/index.php/Information_exposure_through_query_strings_in_url [Accessed 22. Feb. 2019].

[13] Owasp.org. (2019) [online] Available at: https://www.owasp.org/index.php/Top_10-2017_A7-Cross-Site_Scripting_(XSS) [Accessed: 22 Feb. 2019]

[14] Detectify.com (2016) [online] Available at: https://blog.detectify.com/2016/07/13/owasp-top-10-missing-function-level-access-control-7 [Accessed: 23 Feb. 2019]

[15] Github.com [Online] Available at: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md [Accessed: 23 Feb. 2019]

[16] Github.com [online] Accessed on: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Session_Management_Cheat_Sheet.md [Accessed: 23 Feb. 2019]

[17] Github.com [online] Accessed on: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.md[Accessed: 25 Feb. 2019]

[18] Acunetix.com. [online] Accessed on: https://www.acunetix.com/websitesecurity/csrf-attacks/ [Accessed: 25 Feb. 2019]

[19] Qualys.com [online] Accessed on: https://blog.qualys.com/securitylabs/2015/01/14/do-your-anti-csrf-tokens-really-protect-your-applications-from-csrf-attack [Accessed: 25 Feb. 2019]