AUTOMATED TEST REPORT

Table of Contents

Test Report

This report is the result of an automated scan performed with OWASP ZAP on http://localhost/mutillidae, with the purpose of finding out whether the web application was vulnerable. The scan was performed with the help of OWASP ZAP 2.5.0. The report is formatted according to the NIST 800-115 guideline.

Executive Summary

The purpose of this automated test was to confirm whether the target web application meets the most basic security requirements. The test was meant to simulate a malicious attacker exploiting common vulnerabilities and breaches in a web application. The attacks were conducted under the level of access and privilege that a general Internet user would have.

Summary of Results

The automated scan found a total of 80 vulnerabilities. The following types of vulnerabilities were found:

- Cookie No HttpOnly Flag
- Cross Site Scripting (Reflected)
- Directory Browsing
- Password Autocomplete in Browser
- Path Traversal
- Web Browser XSS Protection Not Enabled
- X-Content-Type-Options Header Missing
- X-Frame-Options Header Not Set

These vulnerabilities where found in the following URLs:

- http://localhost/mutillidae/
- http://localhost/mutillidae/index.php?do=toggle-bubble-hints&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/index.php?do=toggle-enforce-ssl&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/index.php?do=toggle-hints&page=%2Fetc%2Fpasswd
- http://localhost/mutillidae/index.php?do=toggle-hints&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/index.php?do=toggle-hints&page=javascript%3Aalert%281%29%3B
- http://localhost/mutillidae/index.php?do=toggle-security&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/index.php?page=%2Fetc%2Fpasswd
- http://localhost/mutillidae/index.php?page=%2Fetc%2Fpasswd&popUpNotificationCode=HPH0
- http://localhost/mutillidae/index.php?page=captured-data.php
- http://localhost/mutillidae/index.php?page=home.php&popUpNotificationCode=HPH0
- http://localhost/mutillidae/index.php?page=javascript%3Aalert%281%29%3B
- http://localhost/mutillidae/index.php?page=javascript%3Aalert%281%29%3B&popUpNotificationCode=HPH0
- http://localhost/mutillidae/index.php?page=login.php
- http://localhost/mutillidae/index.php?page=pen-test-tool-lookup-ajax.php
- http://localhost/mutillidae/index.php?page=pen-test-tool-lookup.php
- http://localhost/mutillidae/index.php?page=register.php
- http://localhost/mutillidae/index.php?page=show-log.php
- http://localhost/mutillidae/index.php?page=sqlmap-targets.php
- http://localhost/mutillidae/index.php?page=user-info.php
- http://localhost/mutillidae/index.php?page=view-someones-blog.php
- http://localhost/mutillidae/webservices/
- http://localhost/mutillidae/webservices/rest/

- http://localhost/mutillidae/webservices/rest/ws-user-account.php
- http://localhost/mutillidae/webservices/rest/ws-user-account.php?username=%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E
- http://localhost/mutillidae/webservices/rest/ws-user-account.php?username=%6a%65%72%65%6d%79%27%20%75%6e%69%6f%6e%20%73%65%6c%65%63%74%20%63%6f%6e%63%61%74%28%27%54%68%65%20%70%61%73%73%77%6f%72%64%20%66%6f%72%20%27%2c%75%73%65%72%6e%61%6d%65%2c%27%20%69%73%20%27%2c%20%70%61%73%73%77%6f%72%64%29%2c%6d%79%73%69%67%6e%61%74%75%72%65%20%66%72%6f%6d%20%61%63%63%6f%75%6e%74%73%20%2d%2d%20
- http://localhost/mutillidae/webservices/rest/ws-user-account.php?username=*
- http://localhost/mutillidae/webservices/rest/ws-user-account.php?username=adrian
- http://localhost/mutillidae/webservices/soap/
- http://localhost/mutillidae/webservices/soap/ws-hello-world.php
- http://localhost/mutillidae/webservices/soap/ws-lookup-dns-record.php
- http://localhost/mutillidae/webservices/soap/ws-user-account.php
- http://localhost/mutillidae/webservices/soap/ws-user-account.php?wsdl
- http://localhost/robots.txt
- http://localhost/sitemap.xml

Attack Narrative

The scan consisted of a first stage of directory browsing (spidering) for a maximum number of 3 children directories. The second stage was an active scan that attempted to execute the following attacks in the target web application:

- Buffer Overflow
- CRLF Injection
- Cross Site Scripting (Persisting and Reflected)
- Format String Error
- Parameter Tempering
- Path Traversal
- Remote File Inclusion
- Remote OS Command Injection
- Server Side Code Injection
- Server Side Include
- SQL Injection

Risk Rating

Vulnerabilities are rated according to their prevalence and their potential impact, as well as the difficulty of the exploit. The total number of vulnerabilities found for each risk level is:

- Low risk: 60
- Medium risk: 13
- Hight risk: 7

Recommendations

Given the vulnerabilities found, some possible recommendations include:

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

For filenames, use stringent whitelists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses, and exclude directory separators such as "/". Use a whitelist of allowable file extensions.

Warning: if you attempt to cleanse your data, then do so that the end result is not in the form that can be dangerous. A sanitizing mechanism can remove characters such as '.' and ';' which may be required for some exploits. An attacker can try to fool the sanitizing mechanism into "cleaning" data into a dangerous form. Suppose the attacker injects a '.' inside a filename (e.g. "sensi.tiveFile") and the sanitizing mechanism removes the character resulting in the valid filename, "sensitiveFile". If the input data are now assumed to be safe, then the file may be compromised.

Inputs should be decoded and canonicalized to the application's current internal representation before being validated. Make sure that your application does not decode the same input twice. Such errors could be used to bypass whitelist schemes by introducing dangerous inputs after they have been checked.

Use a built-in path canonicalization function (such as realpath() in C) that produces the canonical version of the pathname, which effectively removes ".." sequences and symbolic links.

Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.

OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Disable directory browsing. If this is required, make sure the listed files does not induce risks.

Ensure that the HttpOnly flag is set for all cookies.

Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages. If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform MIME-sniffing at all, or that can be directed by the web application/web server to not perform MIME-sniffing.

Ensure that the web browser's XSS filter is enabled, by setting the X-XSS-Protection HTTP response header to '1'.

Most modern Web browsers support the X-Frame-Options HTTP header. Ensure it's set on all web pages returned by your site (if you expect the page to be framed only by pages on your server (e.g. it's part of a FRAMESET) then you'll want to use SAMEORIGIN, otherwise if you never expect the page to be framed, you should use DENY. ALLOW-FROM allows specific websites to frame the web page in supported web browsers).

Phase: Architecture and Design Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies. For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters. Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

Turn off the AUTOCOMPLETE attribute in forms or individual input elements containing password inputs by using AUTOCOMPLETE='OFF'.

Contacting a professional penetration tester to solve these and many more issues is advised.

Appendix: Methodology

It should be noted that this test was performed in an automated manner using tools such as OWASP ZAP, with no direct human intervention. This makes the test non-exhaustive, and it should be assumed that the target web application has many more vulnerabilities and security breaches than the ones mentioned in this report. This test should be taken as a 'lower floor' for possible vulnerabilities. If this automated test is able to find vulnerabilities in a web application, a resourceful human should be able to find many more, most of which would escape the scope of this test. If the tested web application is found to be vulnerable, it is recommended to contact a professional penetration tester to find and mitigate these and other security breaches.