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 2.6.0. 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 106 distinct vulnerabilities. The following types of vulnerabilities were found:

Cookie No HttpOnly Flag

A cookie has been set without the HttpOnly flag, which means that the cookie can be accessed by JavaScript. If a malicious script can be run on this page then the cookie will be accessible and can be transmitted to another site. If this is a session cookie then session hijacking may be possible.

Web Browser XSS Protection Not Enabled

Web Browser XSS Protection is not enabled, or is disabled by the configuration of the 'X-XSS-Protection' HTTP response header on the web server

Password Autocomplete in Browser

The AUTOCOMPLETE attribute is not disabled on an HTML FORM/INPUT element containing password type input. Passwords may be stored in browsers and retrieved.

Application Error Disclosure

This page contains an error/warning message that may disclose sensitive information like the location of the file that produced the unhandled exception. This information can be used to launch further attacks against the web application. The alert could be a false positive if the error message is found inside a documentation page.

Path Traversal

The Path Traversal attack technique allows an attacker access to files, directories, and commands that potentially reside outside the web document root directory. An attacker may manipulate a URL in such a way that the web site will execute or reveal the contents of arbitrary files anywhere on the web server. Any device that exposes an HTTP-based interface is potentially vulnerable to Path Traversal.

Most web sites restrict user access to a specific portion of the file-system, typically called the "web document root" or "CGI root" directory. These directories contain the files intended for user access and the executable necessary to drive web application functionality. To access files or execute commands anywhere on the file-system, Path Traversal attacks will utilize the ability of special-characters sequences.

The most basic Path Traversal attack uses the "../" special-character sequence to alter the resource location requested in the URL. Although most popular web servers will prevent this technique from escaping the web document root, alternate encodings of the "../" sequence may help bypass the security filters. These method variations include valid and invalid Unicode-encoding ("..%u2216" or "..%c0%af") of the forward slash character, backslash characters ("..\") on Windows-based servers, URL encoded characters "%2e%2e%2f"), and double URL encoding ("..%255c") of the backslash character.

Even if the web server properly restricts Path Traversal attempts in the URL path, a web application itself may still be vulnerable due to improper handling of user-supplied input. This is a common problem of web applications that use template mechanisms or load static text from files. In variations of the attack, the original URL parameter value is substituted with the file name of one of the web application's dynamic scripts. Consequently, the results can reveal source code because the file is interpreted as text instead of an executable script. These techniques often employ additional special characters such as the dot (".") to reveal the listing of the current working directory, or "%00" NULL characters in order to bypass rudimentary file extension checks.

X-Content-Type-Options Header Missing

The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy versions of Firefox will use the declared content type (if one is set), rather than performing MIME-sniffing.

X-Frame-Options Header Not Set

X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.

Directory Browsing

It is possible to view the directory listing. Directory listing may reveal hidden scripts, include files , backup source files etc which can be accessed to read sensitive information.

Cross Site Scripting (Reflected)

Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology. When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash. Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

These vulnerabilities were found in the following URLs:

- http://localhost/mutillidae/index.php?do=toggle-hints&page=%2Fetc%2Fpasswd
- http://localhost/mutillidae/webservices/soap/ws-hello-world.php
- http://localhost/mutillidae/index.php?do=toggle-hints&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/index.php?page=%2Fetc%2Fpasswd&popUpNotificationCode=HPH0
- http://localhost/mutillidae/index.php?page=pen-test-tool-lookup.php
- http://localhost/mutillidae/index.php?do=toggle-enforce-ssl&page=home.php
- http://localhost/mutillidae/index.php?page=javascript%3Aalert%281%29%3B
- http://localhost/mutillidae/webservices/soap/ws-user-account.php?wsdl
- http://localhost/mutillidae/index.php?do=toggle-hints&page=home.php
- http://localhost/mutillidae/webservices/

- http://localhost/mutillidae/index.php?do=toggle-bubble-hints&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/index.php?page=show-log.php
- http://localhost/mutillidae/webservices/rest/
- http://localhost/robots.txt
- http://localhost/mutillidae/index.php?page=pen-test-tool-lookup-ajax.php
- http://localhost/mutillidae/index.php?do=toggle-security&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/
- 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=adrian
- http://localhost/mutillidae/index.php?page=view-someones-blog.php
- http://localhost/mutillidae/index.php?page=user-info.php
- http://localhost/mutillidae/index.php?page=add-to-your-blog.php
- http://localhost/mutillidae/webservices/soap/
- http://localhost/mutillidae/index.php?do=toggle-hints&page=javascript%3Aalert%281%29%3B
- http://localhost/mutillidae/index.php?page=sqlmap-targets.php
- http://localhost/mutillidae/index.php?page=register.php
- http://localhost/mutillidae/webservices/soap/ws-lookup-dns-record.php
- http://localhost/mutillidae/index.php?page=home.php&popUpNotificationCode=HPH0
- http://localhost/mutillidae/index.php?page=javascript%3Aalert%281%29%3B&popUpNotificationCode=HPH0
- http://localhost/mutillidae/index.php?do=toggle-security&page=home.php
- http://localhost/mutillidae/index.php?do=toggle-bubble-hints&page=home.php
- http://localhost/sitemap.xml
- http://localhost/mutillidae/webservices/rest/ws-user-account.php?username=*
- http://localhost/mutillidae/webservices/rest/ws-user-account.php
- http://localhost/mutillidae/index.php?page=captured-data.php
- http://localhost/mutillidae/index.php?do=toggle-enforce-ssl&page=/var/www/html/mutillidae/home.php
- http://localhost/mutillidae/index.php?page=%2Fetc%2Fpasswd
- 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/index.php?page=login.php
- http://localhost/mutillidae/webservices/soap/ws-user-account.php

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:

Attack	Description
Buffer Overflow	In the case of a buffer overflow, the extra data might get written into a sensitive memory location. To check for this flaw, the program sends large strings of input text and checks for anomalous behavior.
Client Browser Cache	Checks the headers of secure pages to ensure they don't allow browsers to cache the page.
CRLF Injection	CRLF refers to the Carriage Return and Line Feed special characters used to denote the end of a line. It might be abused by an attacker to split a HTTP response, which can lead to cross-site scripting or cache poisoning . The program tests for this vulnerability by trying to inject a new header into the response by sending a parameter preceded by CRLF characters.
Cross Site Scripting (Reflected)	First, it submits a 'safe' value and checks the response for any instances of this value. Then, for any locations where this value was found in, the program attempts a series of attacks aimed at that location to confirm whether a XSS attack is possible.
Cross Site Scripting (Persistent)	It begins by submitting a 'safe' value much like the reflected variant, but it then spiders the whole application to find any locations where the value might be found and tries to attack them.
Directory Browsing	Tries to gain access to a directory listing by examining the response for patterns commonly used with web server software such as Apache or IIS.
External Redirect	Tries to force a redirect by sending specific parameter values to the URL, and then checks the headers and response to check whether a redirect occurred.
Format String Error	Sends strings of input text that compiled C code would interpret as formatting parameters, and then looks for any anomalies.
Parameter Tempering	Submits requests with parameter values known to cause errors, and compares the response against patterns found in common errors in Java, Microsoft VBScript, PHP and others.
Path Traversal	Tries to access files and directories outside of the web document root using combinations of pathname prefixes and names of local files typically found in Windows or *NIX systems.
Remote File Inclusion	Passes an external URL as a parameter value for the current URL and checks whether this causes the title of the page to change.
Remote OS Command Injection	Submits *NIX and Windows OS commands as URL parameter values that would produce a specific output should the application be vulnerable. If this output isn't found in the response, then it tries a more complex approach that involves blind injection by submitting sleep instructions and timing the responses.
Server Side Code Injection	Submits PHP and ASP attack strings as URL parameter values and then examines the response.
Server Side Include	Detects the OS the server is running on, and then sends a HTML SSI directive. The response is then analyzed to check whether it matches with a pattern indicating the SSI (Server Side Include) was successful.
SQL Injection	Attacks URL parameters and form parameters with valid and invalid SQL syntax, using diverse SQL injection techniques such as error, Boolean or Union based 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: 78
- Medium risk: 21
- Hight risk: 7

Recommendations

Given the vulnerabilities found, some possible recommendations to remediate each of them include:

Cookie No HttpOnly Flag

Ensure that the HttpOnly flag is set for all cookies.

Web Browser XSS Protection Not Enabled

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

Password Autocomplete in Browser

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

Application Error Disclosure

Review the source code of this page. Implement custom error pages. Consider implementing a mechanism to provide a unique error reference/identifier to the client (browser) while logging the details on the server side and not exposing them to the user.

Path Traversal

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.

X-Content-Type-Options Header Missing

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.

X-Frame-Options Header Not Set

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).

Directory Browsing

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

Cross Site Scripting (Reflected)

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.

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.