VULNERABILITY SCANNING REPORT

Attention:

This document contains information that is EXTREMELY CONFIDENTIAL and PRIVILEGED. This information is intended for the sole and private use. By accepting this document, you agree to keep the contents in confidence and not copy, disclose, or distribute this outside your organization.

Contents

1 Introduction

1.1. Overview

This report documents the findings for the Vulnerabilities of http://83.212.174.87. The purpose of engagement is to discover weak links, security updates and provide Solution to against vulnerabilities entitled discovered.
The core objective of this engagement was to assess client's Application against potential known vulnerabilities discovered during the test.

1.2. Scope

The Vulnerability Assessment perform on the following host

- http://83.212.174.87

2 Summary of Finding

The graph below shows a summary of the number of vulnerabilities found for each impact level for the Vulnerabilities Security Assessment. A significant number of high impact vulnerabilities were found that should be addressed as a priority. During the vulnerability security assessment, it was identified that the total of 9 risks were identified, 1 were high, 1 were medium and 7 were Low.

Total Risks	High	Medium	Low
9	1	1	7

Sr. No	Vulnerabilities	Count	Severity
1	Cross Site Scripting (Reflected)	1	High
2	X-Frame-Options Header Not Set	1	Medium
3	Web Browser XSS Protection Not Enabled	1	Low
4	X-Content-Type-Options Header Missing	1	Low
5	SSH Server CBC Mode Ciphers Enabled	1	Low
6	SSH Weak MAC Algorithms Enabled	1	Low
7	Version Disclosure (Apache)	1	Low
8	[Possible] Cross-site Request Forgery in Login Form Detected	1	Low
9	Autocomplete Enabled	1	Low

3 Detailed Findings

Vulnerability	Impact	High
Vulnerability	**Cross Site Scripting (Reflected)**	
Affected IP	83.212.174.87	
Description	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.	

Solution	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.
Vulnerability	Impact **Medium**
Vulnerability	**X-Frame-Options Header Not Set**
Affected IP	83.212.174.87
Description	X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.
Solution	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).
Vulnerability	Impact **Low**
Vulnerability	**Web Browser XSS Protection Not Enabled**
Affected IP	83.212.174.87

Description	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	
Solution	Ensure that the web browser's XSS filter is enabled, by setting the X-XSS-Protection HTTP response header to '1'.	
Vulnerability	Impact	**Low**
Vulnerability	**X-Content-Type-Options Header Missing**	
Affected IP	83.212.174.87	
Description	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.	
Solution	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.	
Vulnerability	Impact	**Low**
Vulnerability	**SSH Server CBC Mode Ciphers Enabled**	
Affected IP	83.212.174.87	
Description	The SSH server is configured to support Cipher Block Chaining (CBC) encryption. This may allow an attacker to recover the plaintext message from the ciphertext. Note that this plugin only checks for the options of the SSH server and does not check for vulnerable software versions.	
Solution	Contact the vendor or consult product documentation to disable CBC mode cipher encryption, and enable CTR or GCM cipher mode encryption.	
Vulnerability	Impact	**Low**
Vulnerability	**SSH Weak MAC Algorithms Enabled**	
Affected IP	83.212.174.87	
Description	The remote SSH server is configured to allow either MD5 or 96-bit MAC algorithms, both of which are considered weak. Note that this plugin only checks for the options of the SSH server, and it does not check for vulnerable software versions.	

Solution	Contact the vendor or consult product documentation to disable MD5 and 96-bit MAC algorithms.	
Vulnerability	Impact	**Low**
Vulnerability	**Version Disclosure (Apache)**	
Affected IP	83.212.174.87	
Description	This information might help an attacker gain a greater understanding of the systems in use and potentially develop further attacks targeted at the specific version of Apache.	
Solution	Configure your web server to prevent information leakage from the SERVER header of its HTTP response.	
Vulnerability	Impact	**Low**
Vulnerability	**[Possible] Cross-site Request Forgery in Login Form Detected**	
Affected IP	83.212.174.87	
Description	CSRF is a very common vulnerability. It's an attack which forces a user to execute unwanted actions on a web application in which the user is currently authenticated. Depending on the application, an attacker can mount any of the actions that can be done by the user such as adding a user, modifying content, deleting data. All the functionality that's available to the victim can be used by the attacker. Only exception to this rule is a page that requires extra information that only the legitimate user can know (such as user's password).	

Solution	Send additional information in each HTTP request that can be used to determine whether the request came from an authorized source. This "validation token" should be hard to guess for attacker who does not already have access to the user's account. If a request is missing a validation token or the token does not match the expected value, the server should reject the request.
	
	If you are posting form in ajax request, custom HTTP headers can be used to prevent CSRF because the browser prevents sites from sending custom HTTP headers to another site but allows sites to send custom HTTP headers to themselves using XMLHttpRequest.
	
	For native XMLHttpRequest (XHR) object in JavaScript;
	
	xhr = new XMLHttpRequest();
	
	xhr.setRequestHeader('custom-header', 'value');	

	For JQuery, if you want to add a custom header (or set of headers) to
	a. individual request
	
	$.ajax({
	url: 'foo/bar',
	headers: { 'x-my-custom-header': 'some value' }
	});
	b. every request
	
	$.ajaxSetup({
	headers: { 'x-my-custom-header': 'some value' }
	});
	
	OR
	
	$.ajaxSetup({
	beforeSend: function(xhr) {
	xhr.setRequestHeader('x-my-custom-header', 'some value');
	}
	});

Vulnerability	Impact **Low**
Vulnerability	**Autocomplete Enabled**
Affected IP	83.212.174.87
Description	Autocomplete is enabled in one or more of the form fields which might contain sensitive information like "username", "credit card" or "CVV". If user chooses to save, data entered in these fields will be cached by the browser. An attacker who can access the victim's browser could steal this information. This is especially important if the application is commonly used in shared computers, such as cyber cafes or airport terminals.

Solution	Add the attribute autocomplete="off" to the form tag or to individual "input" fields. Find all instances of inputs that store private data and disable autocomplete. Fields which contain data such as "Credit Card" or "CCV" type data should not be cached. You can allow the application to cache usernames and remember passwords; however, in most cases this is not recommended. Re-scan the application after addressing the identified issues to ensure all of the fixes have been applied properly