SAMPLE PROJECT

WEB APPLICATION SECURITY SCAN REPORT

CONFIDENTIAL

SCAN PROJECT
SAMPLE PROJECT

WEB APPLICATION URL
http://192.168.1.47/

SCAN STARTED
02 Jul 2020 14:12

SCAN COMPLETED
02 Jul 2020 15:07

CREATED BY
scanrepeat.com

CONFIDENTIALITY STATEMENT

All information contained in this document is provided in commercial confidence for the sole purpose of use by an authorized user in conjunction with ScanRepeat products. The pages of this document shall not be copied, published, or disclosed wholly or in part to any party without ScanRepeat's prior permission in writing, and shall be held in safe custody. These obligations shall not apply to information which is published or becomes known legitimately from some source other than ScanRepeat.

COPYRIGHT INFORMATION

CONTACT INFORMATION

ScanRepeat
a Ventures CDX company

USA
117 Park Avenue
San Jose, CA 95113

EMEA
Laciarska 4
50-104 Wroclaw, Poland

Website: https://scanrepeat.com
Email: contact@scanrepeat.com
Tel: +1 (415) 340-8020

Table of Contents

Executive Summary

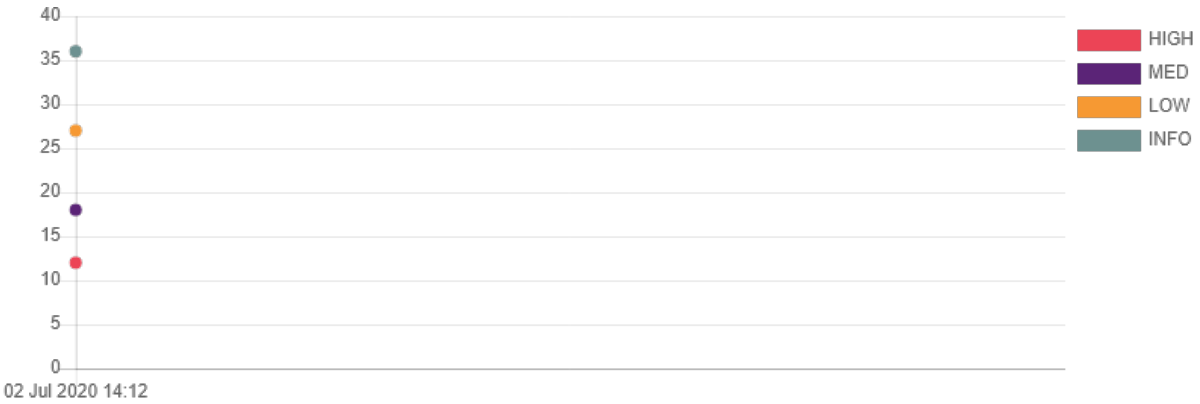
This report documents the results of a web application security scan performed against the web application available at the following URL: **http://192.168.1.47/** by ScanRepeat (https://scanrepeat.com) service that started on 02 Jul 2020 14:12 and completed on 02 Jul 2020 15:07. The option to Scan as a Logged In User was disabled, selected scan type was Active.

The scan results suggest that the tested web application has a **low level** of security in place. The test identified 36 potential security issues including **12 issues of HIGH security risk**.

The summary and timeline of discovered security issues are presented in the sections below. Full details of all issues found with exact locations, descriptions and possible solutions are documented in the “Security Scan Results - Detailed Security Alerts” section.

Security Timeline

The timeline of the potential security issues identified in the last days:



Website: scanrepeat.com Email: contact@scanrepeat.com Tel: +1 (415) 340-8020

Risk Summary

The breakdown of the potential security issues identified during the last security scan:

Total Alerts: 36

Website: scanrepeat.com Email: contact@scanrepeat.com Tel: +1 (415) 340-8020

Risk Scoring

All identified vulnerabilities are classified according to CWE Common Weakness Enumeration ((CWE database)) list of common software security weaknesses and WASC Threat Classification ((WASC database)), and then ranked based on the potential impact and likelihood factors between Informational, Low, Medium and High risk scores.

The assigned risk scoring, whilst provided for informative purposes, could be used to determine the severity and urgency of the reported vulnerabilities.

Risk Score	Description
Informational (INFO)	Informational level issues provide additional insights about your application security features and configuration.
Low (LOW)	Low level issues might not not individually constitute a critical risk these can be used for more advanced attacks when used with each other.
Medium (MEDIUM)	Medium level issues can lead to a very high risk when combined together, these should be resolved with a priority.
High (HIGH)	High level issues individually pose a very high risk and can lead to a very serious impact on data integrity and confidentiality or the availability of the overall application.

Website: scanrepeat.com Email: contact@scanrepeat.com Tel: +1 (415) 340-8020

Alert Definition

The security issues discovered in the scanned web application are reported in detail as security alerts with the following attributes:

Risk Score (XX)	Estimated risk score of the identified issue with the number of occurrences
Alert Name	Vulnerability name
Description	Vulnerability description
Instances	Number of occurrences found
URL / Method	URL and request method (GET/POST) of the vulnerable location
Solution	Possible solution / mitigation
Other Information	Any supplementary information documenting the issue discovery
Reference	External references describing the problem

Open Source Tools Used

ScanRepeat uses the following open source tools as part of its security scanner:

1. OWASP ZAP: https://www.zaproxy.org/docs/
2. RetireJS: https://retirejs.github.io/retire.js/

Website: scanrepeat.com Email: contact@scanrepeat.com Tel: +1 (415) 340-8020

Scan Project Settings

Scan Project:	Sample Project
Scan Frequency:	Daily
Target URL:	http://192.168.1.47/
Scan Type:	Active
Scan Date:	02 Jul 2020 14:12

Website: scanrepeat.com Email: contact@scanrepeat.com Tel: +1 (415) 340-8020

Security Scan Results - Detailed Security Alerts

High(27) - Anti CSRF Tokens Scanner

Description	A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf. CSRF attacks are effective in a number of situations, including: * The victim has an active session on the target site. * The victim is authenticated via HTTP auth on the target site. * The victim is on the same local network as the target site. CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.
Instances	27
URL / Method	POST http://192.168.1.47/vulnerabilities/javascript/ GET http://192.168.1.47/vulnerabilities/xss_d/?default POST http://192.168.1.47/vulnerabilities/exec/ Out of 27 instances
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. For example, use anti-CSRF packages such as the OWASP CSRFGuard. Phase: Implementation Ensure that your application is free of cross-site scripting issues, because most CSRF defenses can be bypassed using attacker-controlled script. Phase: Architecture and Design Generate a unique nonce for each form, place the nonce into the form, and verify the nonce upon receipt of the form. Be sure that the nonce is not predictable (CWE-330). Note that this can be bypassed using XSS. Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation. Note that this can be bypassed using XSS. Use the ESAPI Session Management control. This control includes a component for CSRF. Do not use the GET method for any request that triggers a state change. Phase: Implementation Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.
Other Information	

Reference	http://projects.webappsec.org/Cross-Site-Request-Forgery http://cwe.mitre.org/data/definitions/352.html

High(83) - GDPR/CCPA: Potential exposure of personal identification data (first name and last name)

Description	Phrases that are similar to first name and last name were found on public pages
Instances	83
URL / Method	GET http://192.168.1.47/phpinfo.php GET http://192.168.1.47/instructions.php?doc=readme GET http://192.168.1.47/phpinfo.php Out of 83 instances
Solution	Review and remove potentially exposed personal information.
Other Information	Max Per, Read Me, Daniel R, Jani Taskinen, Libby XML, David Soria, Jerome Loyet, Alain Joye, Peter Cowburn, Pierre Alain, Cross Site, Andrei Zmievski, Lynne Pspell, Antoni Pamies, Johannes Schlüter, Vlad Krupin, Tue May, Nikita Popov, Trace Support, Justin Erenkrantz, Lukas Kahwe, Jason Greene, George Wang, Mac OS, Kristian Koehntopp, Alex Plotnick, Sommer Nielsen, Andi Gutmans, Derick Rethans, Sterling Hughes, Bob Weinand, Andrew Skalski, Boris Lytochkin, Joye XSL, Adam Harvey, Rob Richards, Dan Libby, Ilia Alshanetsky, Stanislav Malyshev, Stig Venaas, Thiago Henrique, Hui Embed, Kalle Sommer, Read Support, Harvey Editor, Henrique Pojda, Core PHP, Brad Dewar, Levi Morrison, Rui Hirokawa, Shane Caraveo, Hui OpenSSL, Ferenc Kovacs, Arnaud Le, Johannes Schlueter, Michael Wallner, Ulf Wendel, Sara Golemon, Christian Cartus, Nick Helm, Tom May, Marcus Boerger, Scott MacVicar, May System, Max Requests, Gregory Beaver, Alex Schoenmaker, Dmitry Stogov, Wendel MySQLnd, Hartmut Holzgraefe
Reference	

High(3) - GDPR/CCPA: Potential exposure of external e-mail adresses

Description	E-mail addresses in external domains were found on public pages
Instances	3
URL / Method	GET http://192.168.1.47/phpinfo.php GET http://192.168.1.47/about.php GET http://192.168.1.47/about.php
Solution	Review and remove e-mail addresses that should not be presented to public

Other Information	license@php.net, contact@scanrepeat.com, team@scanrepeat.com
Reference	

High(4) - Cross Site Scripting (Reflected)

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.
Instances	4
URL / Method	POST http://192.168.1.47/vulnerabilities/xss_s/ POST http://192.168.1.47/vulnerabilities/csp/ POST http://192.168.1.47/vulnerabilities/xss_s/ Out of 4 instances

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.

Other Information	alert(1);, javascript:alert(1);
Reference	http://projects.webappsec.org/Cross-Site-Scripting http://cwe.mitre.org/data/definitions/79.html

High(3) - Remote OS Command Injection

Description	Attack technique used for unauthorized execution of operating system commands. This attack is possible when an application accepts untrusted input to build operating system commands in an insecure manner involving improper data sanitization, and/or improper calling of external programs.
Instances	3
URL / Method	POST http://192.168.1.47/vulnerabilities/captcha/ POST http://192.168.1.47/vulnerabilities/exec/ GET http://192.168.1.47/vulnerabilities/javascript/?query=query%7Ctimeout+%2FT+15
	If at all possible, use library calls rather than external processes to recreate the desired functionality. 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. For any data that will be used to generate a command to be executed, keep as much of that data out of external control as possible. For example, in web applications, this may require storing the command locally in the session's state instead of sending it out to the client in a hidden form field. Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using the ESAPI Encoding control or a similar tool, library, or framework. These will help the programmer encode outputs in a manner less prone to error. If you need to use dynamically-generated query strings or commands in spite of the risk, properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict whitelist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection. If the program to be executed allows arguments to be specified within an input file or from standard input, then consider using that mode to pass arguments instead of the command line. 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. Some

Solution	languages offer multiple functions that can be used to invoke commands. Where possible, identify any function that invokes a command shell using a single string, and replace it with a function that requires individual arguments. These functions typically perform appropriate quoting and filtering of arguments. For example, in C, the system() function accepts a string that contains the entire command to be executed, whereas execl(), execve(), and others require an array of strings, one for each argument. In Windows, CreateProcess() only accepts one command at a time. In Perl, if system() is provided with an array of arguments, then it will quote each of the arguments. 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." When constructing OS command strings, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. This will indirectly limit the scope of an attack, but this technique is less important than proper output encoding and escaping. Note that proper output encoding, escaping, and quoting is the most effective solution for preventing OS command injection, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent OS command injection, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, when invoking a mail program, you might need to allow the subject field to contain otherwise-dangerous inputs like ";" and ">" characters, which would need to be escaped or otherwise handled. In this case, stripping the character might reduce the risk of OS command injection, but it would produce incorrect behavior because the subject field would not be recorded as the user intended. This might seem to be a minor inconvenience, but it could be more important when the program relies on well-structured subject lines in order to pass messages to other components. Even if you make a mistake in your validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address.
Other Information	
Reference	http://cwe.mitre.org/data/definitions/78.html https://owasp.org/www-community/attacks/Command_Injection

High(5) - SQL Injection

Description	SQL injection may be possible.
Instances	5
URL / Method	POST http://192.168.1.47/vulnerabilities/xss_s/ POST http://192.168.1.47/vulnerabilities/upload/ POST http://192.168.1.47/vulnerabilities/xss_s/?query=query+AND+1%3D1 Out of 5 instances
Solution	Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side. If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?' If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries. If database Stored Procedures can be used, use them. Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality! Do not create dynamic SQL queries using simple string concatenation. Escape all data received from the client. Apply a 'whitelist' of allowed characters, or a 'blacklist' of disallowed characters in user input. Apply the principle of least privilege by using the least privileged database user possible. In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact. Grant the minimum database access that is necessary for the application.
Other Information	The page results were successfully manipulated using the boolean conditions [ZAP AND 1=1] and [ZAP AND 1=2] The parameter value being modified was stripped from the HTML output for the purposes of the comparison Data was returned for the original parameter. The vulnerability was detected by successfully restricting the data originally returned, by manipulating the parameter
Reference	https://cheatsheetseries.owasp.org/cheatsheets /SQL_Injection_Prevention_Cheat_Sheet.html

High(2) - PII Disclosure

Description	The response contains Personally Identifiable Information, such as CC number, SSN and similar sensitive data.
Instances	2
URL / Method	GET http://192.168.1.47/about.php GET http://192.168.1.47/about.php
Solution	

Other Information	Credit Card Type detected: Visa Bank Identification Number: 400000 Brand: VISA Category: Issuer:
Reference	

High(5) - Cross Site Scripting (DOM Based)

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.
Instances	5

URL / Method	GET http://192.168.1.47/vulnerabilities/xss_d/?default#alert(1)
	GET http://192.168.1.47/vulnerabilities/xss_s/#jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert())//%0D%0A%0d%0a//\x3csVg/\x3e
	POST http://192.168.1.47/vulnerabilities/xss_s/#jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert())//%0D%0A%0d%0a//\x3csVg/\x3e
	Out of 5 instances

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.

Other Information	Tag name: input Att name: Att id:
Reference	http://projects.webappsec.org/Cross-Site-Scripting http://cwe.mitre.org/data/definitions/79.html

High(2) - Cross Site Scripting (Reflected)

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.
Instances	2
URL / Method	GET http://192.168.1.47/vulnerabilities/brute/?Login=Login&password=ZAP&username=%27%22%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E GET http://192.168.1.47/vulnerabilities/sqli/?Submit=Submit&id=%27%22%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E

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.

Other Information	'"alert(1);
Reference	http://projects.webappsec.org/Cross-Site-Scripting http://cwe.mitre.org/data/definitions/79.html

High(1) - Path Traversal

Description	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.
Instances	1
URL / Method	GET http://192.168.1.47/vulnerabilities/fi/?page=%2Fetc%2Fpasswd

Solution	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.
Other Information	root:x:0:0
Reference	http://projects.webappsec.org/Path-Traversal http://cwe.mitre.org/data/definitions/22.html

High(1) - GDPR/CCPA: Possible detection of exposed France IBAN Bank Account number

Description	France Iban number was found
Instances	1
URL / Method	GET http://192.168.1.47/about.php
Solution	Review and remove wrongly exposed IBAN Bank Account numbers
Other Information	FR76 3000 6000 0112 3456 7890 189
Reference	

High(1) - GDPR/CCPA: Possible detection of exposed Germany IBAN Bank Account number

Description	Germany Iban number was found
Instances	1
URL / Method	GET http://192.168.1.47/about.php
Solution	Review and remove wrongly exposed IBAN Bank Account numbers
Other Information	DE91 1000 0000 0123 4567 89
Reference	

Medium(1) - HTTP Only Site

Description	The site is only served under HTTP and not HTTPS.
Instances	1
URL / Method	GET http://192.168.1.47/
Solution	Configure your web or application server to use SSL (https).
Other Information	Failed to connect. ZAP attempted to connect via: https://192.168.1.47:443/
Reference	https://cheatsheetseries.owasp.org/cheatsheets /Transport_Layer_Protection_Cheat_Sheet.html

	https://letsencrypt.org/

Medium(37) - Reverse Tabnabbing

Description	At least one link on this page is vulnerable to Reverse tabnabbing as it uses a target attribute without using both of the "noopener" and "noreferrer" keywords in the "rel" attribute, which allows the target page to take control of this page.
Instances	37
URL / Method	GET http://192.168.1.47/instructions.php?doc=readme GET http://192.168.1.47/vulnerabilities/brute/ GET http://192.168.1.47/vulnerabilities/fi/?page=file3.php Out of 37 instances
Solution	Do not use a target attribute, or if you have to then also add the attribute: rel="noopener noreferrer".
Other Information	https://www.virtualbox.org/, https://www.owasp.org/index.php/Testing_for_Brute_Force_(OWASP-AT-004), https://en.wikipedia.org/wiki/Remote_File_Inclusion, https://www.owasp.org/index.php/Cross-site_Scripting_(XSS), https://www.owasp.org/index.php/Unrestricted_File_Upload, http://fsf.org/, http://hiderefer.com, Damn Vulnerable Web Application (DVWA), https://www.owasp.org/index.php/Cross-Site_Request_Forgery, http://www.securiteam.com/securityreviews/5DP0N1P76E.html, http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution, Content Security Policy Reference, https://www.w3schools.com/js/, https://www.google.com/recaptcha/admin/create, PHPIDS, http://www.dvwa.co.uk/, VirtualBox
Reference	https://owasp.org/www-community/attacks/Reverse_Tabnabbing https://dev.to/ben/the-targetblank-vulnerability-by-example https://mathiasbynens.github.io/rel-noopener/ https://medium.com/@jitbit/target-blank-the-most-underestimated-vulnerability-ever-96e328301f4c

Medium(10) - Relative Path Confusion

Description	The web server is configured to serve responses to ambiguous URLs in a manner that is likely to lead to confusion about the correct "relative path" for the URL. Resources (CSS, images, etc.) are also specified in the page response using relative, rather than absolute URLs. In an attack, if the web browser parses the "cross-content" response in a permissive manner, or can be tricked into permissively parsing the "cross-content" response, using techniques such as framing, then the web browser may be fooled into interpreting HTML as CSS (or other content types), leading to an XSS vulnerability.
Instances	10

URL / Method	GET http://192.168.1.47/login.php GET http://192.168.1.47/instructions.php?doc=PHPIDS-license GET http://192.168.1.47/security.php?phpids=on Out of 10 instances
Solution	Web servers and frameworks should be updated to be configured to not serve responses to ambiguous URLs in such a way that the relative path of such URLs could be mis-interpreted by components on either the client side, or server side. Within the application, the correct use of the "" HTML tag in the HTTP response will unambiguously specify the base URL for all relative URLs in the document. Use the "Content-Type" HTTP response header to make it harder for the attacker to force the web browser to mis-interpret the content type of the response. Use the "X-Content-Type-Options: nosniff" HTTP response header to prevent the web browser from "sniffing" the content type of the response. Use a modern DOCTYPE such as "" to prevent the page from being rendered in the web browser using "Quirks Mode", since this results in the content type being ignored by the web browser. Specify the "X-Frame-Options" HTTP response header to prevent Quirks Mode from being enabled in the web browser using framing attacks.
Other Information	No tag was specified in the HTML tag to define the location for relative URLs. A Content Type of "text/html;charset=utf-8" was specified. If the web browser is employing strict parsing rules, this will prevent cross-content attacks from succeeding. Quirks Mode in the web browser would disable strict parsing. Quirks Mode is implicitly enabled via the use of an old DOCTYPE with PUBLIC id "-//W3C//DTD XHTML 1.0 Strict//EN", allowing the specified Content Type to be bypassed in some web browsers.
Reference	http://www.thespanner.co.uk/2014/03/21/rpo/ https://hsivonen.fi/doctype/ http://www.w3schools.com/tags/tag_base.asp

Medium(2) - CSP Scanner: Wildcard Directive

Description	The following directives either allow wildcard sources (or ancestors), are not defined, or are overly broadly defined: style-src, style-src-elem, style-src-attr, img-src, connect-src, frame-src, frame-ancestors, font-src, media-src, object-src, manifest-src, prefetch-src
Instances	2
URL / Method	GET http://192.168.1.47/vulnerabilities/csp/ POST http://192.168.1.47/vulnerabilities/csp/
Solution	Ensure that your web server, application server, load balancer, etc. is properly configured to set the Content-Security-Policy header.
Other Information	script-src 'self' https://pastebin.com example.com code.jquery.com https://ssl.google-analytics.com ;

Reference	http://www.w3.org/TR/CSP2/ http://www.w3.org/TR/CSP/ http://caniuse.com/#search=content+security+policy http://content-security-policy.com/ https://github.com/shapesecurity/salvation

Medium(43) - X-Frame-Options Header Not Set

Description	X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.
Instances	43
URL / Method	GET http://192.168.1.47/vulnerabilities/csrf/?Change=Change&password_conf=ZAP&password_new=ZAP GET http://192.168.1.47/vulnerabilities/xss_r/?name=ZAP GET http://192.168.1.47/instructions.php?doc=copying Out of 43 instances
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).
Other Information	
Reference	https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options

Medium(6) - Directory Browsing

Description	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.
Instances	6
URL / Method	GET http://192.168.1.47/docs/ GET http://192.168.1.47/vulnerabilities/ GET http://192.168.1.47/dvwa/images/ Out of 6 instances

Solution	Disable directory browsing. If this is required, make sure the listed files does not induce risks.
Other Information	
Reference	http://httpd.apache.org/docs/mod/core.html#options http://alamo.satlug.org/pipermail/satlug/2002-February/000053.html

Low(44) - Content Security Policy (CSP) Header Not Set

Description	Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware. CSP provides a set of standard HTTP headers that allow website owners to declare approved sources of content that browsers should be allowed to load on that page — covered types are JavaScript, CSS, HTML frames, fonts, images and embeddable objects such as Java applets, ActiveX, audio and video files.
Instances	44
URL / Method	GET http://192.168.1.47/vulnerabilities/xss_d/?default GET http://192.168.1.47/instructions.php?doc=PHPIDS-license GET http://192.168.1.47/setup.php Out of 44 instances
Solution	Ensure that your web server, application server, load balancer, etc. is configured to set the Content-Security-Policy header, to achieve optimal browser support: "Content-Security-Policy" for Chrome 25+, Firefox 23+ and Safari 7+, "X-Content-Security-Policy" for Firefox 4.0+ and Internet Explorer 10+, and "X-WebKit-CSP" for Chrome 14+ and Safari 6+.
Other Information	
Reference	https://developer.mozilla.org/en-US/docs/Web/Security/CSP /Introducing_Content_Security_Policy https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html http://www.w3.org/TR/CSP/ http://w3c.github.io/webappsec/specs/content-security-policy/csp-specification.dev.html http://www.html5rocks.com/en/tutorials/security/content-security-policy/ http://caniuse.com/#feat=contentsecuritypolicy http://content-security-policy.com/

Low(64) - Server Leaks Version Information via "Server" HTTP Response Header Field

Description	The web/application server is leaking version information via the "Server" HTTP response header. Access to such information may facilitate attackers identifying other vulnerabilities your web/application server is subject to.
Instances	64
URL / Method	GET http://192.168.1.47/dvwa/images/login_logo.png GET http://192.168.1.47/sitemap.xml GET http://192.168.1.47/dvwa/js/dvwaPage.js Out of 64 instances
Solution	Ensure that your web server, application server, load balancer, etc. is configured to suppress the "Server" header or provide generic details.
Other Information	Apache/2.4.25 (Debian)
Reference	http://httpd.apache.org/docs/current/mod/core.html#servertokens http://msdn.microsoft.com/en-us/library/ff648552.aspx#ht_urlscan_007 http://blogs.msdn.com/b/varunm/archive/2013/04/23/remove-unwanted-http-response-headers.aspx http://www.troyhunt.com/2012/02/shhh-dont-let-your-response-headers.html

Low(30) - Absence of Anti-CSRF Tokens

Description	No Anti-CSRF tokens were found in a HTML submission form. A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf. CSRF attacks are effective in a number of situations, including: * The victim has an active session on the target site. * The victim is authenticated via HTTP auth on the target site. * The victim is on the same local network as the target site. CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.
Instances	30
	GET http://192.168.1.47/vulnerabilities/xss_d/

URL / Method	GET http://192.168.1.47/vulnerabilities/csrf/?Change=Change&password_conf=ZAP&password_new=ZAP GET http://192.168.1.47/login.php Out of 30 instances
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. For example, use anti-CSRF packages such as the OWASP CSRFGuard. Phase: Implementation Ensure that your application is free of cross-site scripting issues, because most CSRF defenses can be bypassed using attacker-controlled script. Phase: Architecture and Design Generate a unique nonce for each form, place the nonce into the form, and verify the nonce upon receipt of the form. Be sure that the nonce is not predictable (CWE-330). Note that this can be bypassed using XSS. Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation. Note that this can be bypassed using XSS. Use the ESAPI Session Management control. This control includes a component for CSRF. Do not use the GET method for any request that triggers a state change. Phase: Implementation Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.
Other Information	No known Anti-CSRF token [anticsrf, CSRFToken, __RequestVerificationToken, csrfmiddlewaretoken, authenticity_token, OWASP_CSRFTOKEN, anoncsrf, csrf_token, _csrf, _csrfSecret, uset_token, csrf, X-CSRF_TOKEN] was found in the following HTML form: [Form 1:].
Reference	http://projects.webappsec.org/Cross-Site-Request-Forgery http://cwe.mitre.org/data/definitions/352.html

Low(55) - X-Content-Type-Options Header Missing

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.
Instances	55
URL / Method	GET http://192.168.1.47/favicon.ico GET http://192.168.1.47/vulnerabilities/fi/?page=file1.php GET http://192.168.1.47/vulnerabilities/xss_r/?name=ZAP Out of 55 instances

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.
Other Information	This issue still applies to error type pages (401, 403, 500, etc.) as those pages are often still affected by injection issues, in which case there is still concern for browsers sniffing pages away from their actual content type. At "High" threshold this scanner will not alert on client or server error responses.
Reference	http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx https://owasp.org/www-community/Security_Headers

Low(2) - Cross-Domain JavaScript Source File Inclusion

Description	The page includes one or more script files from a third-party domain.
Instances	2
URL / Method	POST http://192.168.1.47/vulnerabilities/captcha/ GET http://192.168.1.47/vulnerabilities/captcha/
Solution	Ensure JavaScript source files are loaded from only trusted sources, and the sources can't be controlled by end users of the application.
Other Information	
Reference	

Low(4) - Private IP Disclosure

Description	A private IP (such as 10.x.x.x, 172.x.x.x, 192.168.x.x) or an Amazon EC2 private hostname (for example, ip-10-0-56-78) has been found in the HTTP response body. This information might be helpful for further attacks targeting internal systems.
Instances	4
URL / Method	GET http://192.168.1.47/vulnerabilities/fi/?page=file3.php GET http://192.168.1.47/ids_log.php GET http://192.168.1.47/vulnerabilities/fi/?page=file1.php Out of 4 instances

Solution	Remove the private IP address from the HTTP response body. For comments, use JSP /ASP/PHP comment instead of HTML/JavaScript comment which can be seen by client browsers.
Other Information	172.17.0.1
Reference	https://tools.ietf.org/html/rfc1918

Low(5) - Cookie No HttpOnly Flag

Description	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.
Instances	5
URL / Method	POST http://192.168.1.47/vulnerabilities/weak_id/ POST http://192.168.1.47/security.php POST http://192.168.1.47/security.php Out of 5 instances
Solution	Ensure that the HttpOnly flag is set for all cookies.
Other Information	Set-Cookie: dvwaSession, Set-Cookie: security, Set-Cookie: PHPSESSID
Reference	https://owasp.org/www-community/HttpOnly

Low(5) - Cookie Without SameSite Attribute

Description	A cookie has been set without the SameSite attribute, which means that the cookie can be sent as a result of a 'cross-site' request. The SameSite attribute is an effective counter measure to cross-site request forgery, cross-site script inclusion, and timing attacks.
Instances	5
URL / Method	POST http://192.168.1.47/security.php GET http://192.168.1.47/ POST http://192.168.1.47/vulnerabilities/weak_id/ Out of 5 instances
Solution	Ensure that the SameSite attribute is set to either 'lax' or ideally 'strict' for all cookies.
Other Information	Set-Cookie: security, Set-Cookie: dvwaSession, Set-Cookie: PHPSESSID

Reference	https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site

Low(2) - Information Disclosure - Debug Error Messages

Description	The response appeared to contain common error messages returned by platforms such as ASP.NET, and Web-servers such as IIS and Apache. You can configure the list of common debug messages.
Instances	2
URL / Method	GET http://192.168.1.47/instructions.php?doc=readme GET http://192.168.1.47/instructions.php
Solution	Disable debugging messages before pushing to production.
Other Information	PHP warning
Reference	

Info(100) - Timestamp Disclosure - Unix

Description	A timestamp was disclosed by the application/web server - Unix
Instances	100
URL / Method	POST http://192.168.1.47/vulnerabilities/javascript/ GET http://192.168.1.47/vulnerabilities/javascript/ POST http://192.168.1.47/vulnerabilities/javascript/ Out of 100 instances
Solution	Manually confirm that the timestamp data is not sensitive, and that the data cannot be aggregated to disclose exploitable patterns.
Other Information	606105819, which evaluates to: 1989-03-17 03:43:39
Reference	http://projects.webappsec.org/w/page/13246936/Information%20Leakage

Info(100) - User Agent Fuzzer

Description	Check for differences in response based on fuzzed User Agent (eg. mobile sites, access as a Search Engine Crawler). Compares the response statuscode and the hashcode of the response body with the original response.
Instances	100
URL / Method	GET http://192.168.1.47/logout.php GET http://192.168.1.47/vulnerabilities/captcha/ GET http://192.168.1.47/login.php Out of 100 instances
Solution	
Other Information	
Reference	https://owasp.org/wstg

Info(4) - Information Disclosure - Sensitive Information in URL

Description	The request appeared to contain sensitive information leaked in the URL. This can violate PCI and most organizational compliance policies. You can configure the list of strings for this check to add or remove values specific to your environment.
Instances	4
URL / Method	GET http://192.168.1.47/vulnerabilities/brute/?Login=Login&password=ZAP&username=ZAP GET http://192.168.1.47/vulnerabilities/csrf/?Change=Change&password_conf=ZAP&password_new=ZAP GET http://192.168.1.47/vulnerabilities/brute/?Login=Login&password=ZAP&username=ZAP Out of 4 instances
Solution	Do not pass sensitive information in URIs.
Other Information	The URL contains potentially sensitive information. The following string was found via the pattern: user username
Reference	

Info(63) - Cookie Slack Detector

Description	Repeated GET requests: drop a different cookie each time, followed by normal request with all cookies to stabilize session, compare responses against original baseline GET. This can reveal areas where cookie based authentication/attributes are not actually enforced.
Instances	63
URL / Method	GET http://192.168.1.47/dvwa/images/lock.png GET http://192.168.1.47/vulnerabilities/sqli_blind/ GET http://192.168.1.47/vulnerabilities/csp/ Out of 63 instances
Solution	
Other Information	Cookies that don't have expected effects can reveal flaws in application logic. In the worst case, this can reveal where authentication via cookie token(s) is not actually enforced. These cookies affected the response: These cookies did NOT affect the response: security,PHPSESSID
Reference	http://projects.webappsec.org/Fingerprinting

Info(20) - User Controllable HTML Element Attribute (Potential XSS)

Description	This check looks at user-supplied input in query string parameters and POST data to identify where certain HTML attribute values might be controlled. This provides hot-spot detection for XSS (cross-site scripting) that will require further review by a security analyst to determine exploitability.
Instances	20
URL / Method	POST http://192.168.1.47/vulnerabilities/javascript/ POST http://192.168.1.47/vulnerabilities/upload/ POST http://192.168.1.47/vulnerabilities/csp/ Out of 20 instances
Solution	Validate all input and sanitize output it before writing to any HTML attributes.
Other Information	User-controlled HTML attribute values were found. Try injecting special characters to see if XSS might be possible. The page at the following URL: http://192.168.1.47 /vulnerabilities/javascript/ appears to include user input in: a(n) [input] tag [value] attribute The user input found was: send=Submit The user-controlled value was: submit
Reference	http://websecuritytool.codeplex.com/wikipage?title=Checks#user-controlled-html-attribute

Info(1) - Information Disclosure - Suspicious Comments

Description	The response appears to contain suspicious comments which may help an attacker. Note: Matches made within script blocks or files are against the entire content not only comments.
Instances	1
URL / Method	GET http://192.168.1.47/setup.php
Solution	Remove all comments that return information that may help an attacker and fix any underlying problems they refer to.
Other Information	The following comment/snippet was identified via the pattern: \bDB\b
Reference	

Info(2) - Information Disclosure - Suspicious Comments

Description	The response appears to contain suspicious comments which may help an attacker. Note: Matches made within script blocks or files are against the entire content not only comments.
Instances	2
URL / Method	GET http://192.168.1.47/vulnerabilities/javascript/ POST http://192.168.1.47/vulnerabilities/javascript/
Solution	Remove all comments that return information that may help an attacker and fix any underlying problems they refer to.
Other Information	The following comment/snippet was identified via the pattern: \bFROM\b /* MD5 code from here https://github.com/blueimp/JavaScript-MD5 */ !function(n){"use strict";function t(n,t){var r=(65535&n)+(65535&t);return(n>>16)+(t>>16)+(r>>16)>>32-t}function e(n,e,o,u,c,f){return t(r(t(t(e,n),t(u,f)),c),o)}function o(n,t,r,o,u,c,f){return e(t&r|~t&o,n,t,u,c,f)}function u(n,t,r,o,u,c,f){return e(t&o|r&~o,n,t,u,c,f)}function c(n,t,r,o,u,c,f){return e(t^r^o,n,t,u,c,f)}function f(n,t,r,o,u,c,f){return e(r^(t|~o),n,t,u,c,f)}function i(n,r){n[r>>5]|=128>>9>5]>>>t%32&255);return r}function d(n){var t,r=[];for(r[(n.length>>2)-1]=void 0,t=0;t>5]|=(255&n.charCodeAt(t/8))16&&(o=i(o,8*n.length)),r=0;r>>4&15)+"0123456789abcdef".charAt(15&t);return e}function v(n){return unescape(encodeURIComponent(n))}function m(n){return h(v(n))}function p(n){return g(m(n))}function s(n,t){return l(v(n),v(t))}function C(n,t){return g(s(n,t))}function A(n,t,r){return t?r?s(t,n):C(t,n):r?m(n):p(n)}"function"==typeof define&&define.amd?define(function(){return A}):"object"==typeof module&&module.exports?module.exports=A:n.md5=A}(this); function rot13(inp) { return inp.replace(/[a-zA-Z]/g,function(c){return String.fromCharCode((c=(c=c.charCodeAt(0)

	+13)?c:c-26);}); } function generate_token() { var phrase = document.getElementById ("phrase").value; document.getElementById("token").value = md5(rot13(phrase)); } generate_token();
Reference	

Info(1) - Cookie Poisoning

Description	This check looks at user-supplied input in query string parameters and POST data to identify where cookie parameters might be controlled. This is called a cookie poisoning attack, and becomes exploitable when an attacker can manipulate the cookie in various ways. In some cases this will not be exploitable, however, allowing URL parameters to set cookie values is generally considered a bug.
Instances	1
URL / Method	POST http://192.168.1.47/security.php
Solution	Do not allow user input to control cookie names and values. If some query string parameters must be set in cookie values, be sure to filter out semicolon's that can serve as name/value pair delimiters.
Other Information	An attacker may be able to poison cookie values through POST parameters. To test if this is a more serious issue, you should try resending that request as a GET, with the POST parameter included as a query string parameter. For example: http://nottrusted.com/page?value=maliciousInput. This was identified at: http://192.168.1.47/security.php User-input was found in the following cookie: security=low The user input was: security=low
Reference	http://websecuritytool.codeplex.com/wikipage?title=Checks#user-controlled-cookie

Info(1) - Modern Web Application

Description	The application appears to be a modern web application. If you need to explore it automatically then the Ajax Spider may well be more effective than the standard one.
Instances	1
URL / Method	GET http://192.168.1.47/phpinfo.php
Solution	This is an informational alert and so no changes are required.
Other Information	Links have been found that do not have traditional href attributes, which is an indication that this is a modern web application.

Reference

Website: scanrepeat.com Email: contact@scanrepeat.com Tel: +1 (415) 340-8020

Tested URL Resources

The following resources were analyzed during the scan.

http://192.168.1.47/dvwa

https://192.168.1.47

http://192.168.1.47/

http://192.168.1.47/about.php

http://192.168.1.47/about.php/c9a7n

http://192.168.1.47/about.php/c9a7n/84zqc

http://192.168.1.47/docs

http://192.168.1.47/docs/DVWA_v1.3.pdf

http://192.168.1.47/docs/

http://192.168.1.47/dvwa/css

http://192.168.1.47/dvwa/css/login.css

http://192.168.1.47/dvwa/css/main.css

http://192.168.1.47/dvwa/css/

http://192.168.1.47/dvwa/images/

http://192.168.1.47/dvwa/images

http://192.168.1.47/dvwa/images/lock.png

http://192.168.1.47/dvwa/images/login_logo.png

http://192.168.1.47/dvwa/images/logo.png

http://192.168.1.47/dvwa/images/RandomStorm.png

http://192.168.1.47/dvwa/images/spanner.png

http://192.168.1.47/dvwa/js/

http://192.168.1.47/dvwa/js

http://192.168.1.47/dvwa/js/add_event_listeners.js

http://192.168.1.47/dvwa/js/dvwaPage.js

http://192.168.1.47/dvwa/

http://192.168.1.47/favicon.ico

http://192.168.1.47/ids_log.php

http://192.168.1.47/ids_log.php/c9a7n

http://192.168.1.47/ids_log.php/c9a7n/84zqc

http://192.168.1.47/ids_log.php/c9a7n/84zqc?clear_log=Clear%20Log

http://192.168.1.47/ids_log.php?clear_log=Clear+Log

http://192.168.1.47/instructions.php

http://192.168.1.47/instructions.php/c9a7n

http://192.168.1.47/instructions.php/c9a7n/84zqc

http://192.168.1.47/instructions.php/c9a7n/84zqc?doc=PHPIDS-license

http://192.168.1.47/instructions.php?doc=PHPIDS-license

http://192.168.1.47/login.php

http://192.168.1.47/login.php/c9a7n

http://192.168.1.47/login.php/c9a7n/84zqc

http://192.168.1.47/logout.php

http://192.168.1.47/phpinfo.php

http://192.168.1.47/phpinfo.php/c9a7n

http://192.168.1.47/phpinfo.php/c9a7n/84zqc

http://192.168.1.47/robots.txt

http://192.168.1.47/security.php

http://192.168.1.47/security.php/c9a7n

http://192.168.1.47/security.php/c9a7n/84zqc

http://192.168.1.47/security.php/c9a7n/84zqc?phpids=on

http://192.168.1.47/security.php?phpids=on

http://192.168.1.47/security.php?test=%2522%3E%3Cscript%3Eeval(window.name)%3C/script%3E

http://192.168.1.47/setup.php

http://192.168.1.47/setup.php/c9a7n

http://192.168.1.47/setup.php/c9a7n/84zqc

http://192.168.1.47/sitemap.xml

http://192.168.1.47/vulnerabilities/

http://192.168.1.47/vulnerabilities

http://192.168.1.47/vulnerabilities/brute/

http://192.168.1.47/vulnerabilities/brute/?Login=Login&password=ZAP&username=ZAP

http://192.168.1.47/vulnerabilities/captcha/

http://192.168.1.47/vulnerabilities/csp

http://192.168.1.47/vulnerabilities/csp/ZAP

http://192.168.1.47/vulnerabilities/csp/

http://192.168.1.47/vulnerabilities/csrf/

http://192.168.1.47/vulnerabilities/csrf/?Change=Change&password_conf=ZAP&password_new=ZAP

http://192.168.1.47/vulnerabilities/exec/

http://192.168.1.47/vulnerabilities/fi/?page=file3.php

http://192.168.1.47/vulnerabilities/javascript/

http://192.168.1.47/vulnerabilities/javascript/?query=query%7Ctimeout+%2FT+15

http://192.168.1.47/vulnerabilities/sqli/

http://192.168.1.47/vulnerabilities/sqli/?Submit=Submit&id=ZAP

http://192.168.1.47/vulnerabilities/sqli_blind/

http://192.168.1.47/vulnerabilities/sqli_blind/?Submit=Submit&id=ZAP

http://192.168.1.47/vulnerabilities/upload/

http://192.168.1.47/vulnerabilities/weak_id/

http://192.168.1.47/vulnerabilities/xss_d/

http://192.168.1.47/vulnerabilities/xss_d/?default

http://192.168.1.47/vulnerabilities/xss_r/

http://192.168.1.47/vulnerabilities/xss_r/?name=ZAP

http://192.168.1.47/vulnerabilities/xss_s/

http://192.168.1.47/vulnerabilities/xss_s/?query=query+AND+1%3D1

Website: scanrepeat.com Email: contact@scanrepeat.com Tel: +1 (415) 340-8020

CONTACT

Website: scanrepeat.com
Email: contact@scanrepeat.com
Tel: +1 (415) 340-8020