**Vulnerability Assessment Report**

**For**



**burpresult3**

**March 22, 2022**

**Document Security Level:** Confidential

**Document Version:** 1.0

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Edit Report** | **Editor** |
| 1.0 | March 22, 2022 | Creation | INET Managed Services |

|  |  |  |  |
| --- | --- | --- | --- |
| burpresult3 | | INET Managed Services CO., LTD. | |
| Name |  | **Name** |  |
| Position |  | **Position** |  |
| Tel |  | **Tel** |  |
| Signature |  | **Signature** |  |

**Table of Contents**

[1. Restrictions on disclosure and use of information 4](#_Toc98504828)

[2. Operation Method 5](#_Toc98504829)

[3. Project Scope 6](#_Toc98504830)

[3.1 Web Application Vulnerability Assessment 6](#_Toc98504831)

[4. Testing Tools 7](#_Toc98504832)

[5. Web Application Vulnerability Assessment 7](#_Toc98504833)

[6.1 Target Information 8](#_Toc98504834)

[6.2 Executive summary 9](#_Toc98504835)

[6.2.1 Summary Vulnerability by Severity 9](#_Toc98504836)

[6.2.2 Vulnerability by Target 9](#_Toc98504837)

[6.3 Web Application Vulnerability Detail 10](#_Toc98504838)

[6. Port Discovery 11](#_Toc98504839)

[7. Appendix 12](#_Toc98504840)

[8.1 About Nessus 12](#_Toc98504841)

[8.1.1 Nessus vulnerabilities 12](#_Toc98504842)

[8.1.2 Nessus risk score 12](#_Toc98504843)

[8.2 About Burp Suite's web vulnerability scanner 14](#_Toc98504844)

[8.2.1 Burp Suite's web vulnerability scanner risk score 14](#_Toc98504845)

# Restrictions on disclosure and use of information

Restriction on Disclosure and Use of Confidential Information. The Executive understands and agrees that the Confidential Information constitutes an asset of the Company and its affiliated entities and may not be converted to the Executive's own use. Accordingly, the Executive hereby agrees that the Executive shall not, directly, or indirectly, at any time, reveal, divulge, or disclose to any Person not expressly authorized by the Company any Confidential Information, and the Executive shall not, directly, or indirectly, use or make use of any Confidential Information in connection with any business activity other than that of the Company. The parties acknowledge and agree that this Agreement is not intended to, and does not, alter either the Company's rights or the Executive's obligations under any state or federal statutory or common law regarding trade secrets and unfair trade.

# Operation Method

* 1. Posture Review
  2. Information Gathering
  3. Enumeration
  4. Vulnerability Assessment
  5. Analyze & Evaluate Risk Value
  6. Report



Figure 1: Operation Method

# Project Scope

## **3.1 Web Application Vulnerability Assessment**

**Target / IP Address:**

| **No.** | **Domain / Server Name** | **Public IP Address** | **Private IP Address** | **OS/Model** | **Functions** | **Public Assessment** | **Private Assessment** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | - | - | - | - | - | ✓ | - |

# Testing Tools

|  |  |
| --- | --- |
| **Tool Name** | **Testing Type** |
| Nmap | Host and Service Discovery |
| Nessus Professional | Infrastructure Vulnerability Assessment |
| Burp Suite's web vulnerability scanner | Web Application Vulnerability Assessment |

# Web Application Vulnerability Assessment

**Vulnerability Assessment from Public Access (for public target)**

**Testing date:** <<Date SCAN>>

**Tester IP Address:** <<IP Target>>

Diagram

Description automatically generated

Figure 5: Vulnerability Assessment from Public Access

## **5.1 Target Information**

| **No.** | **Domain / Server Name** | **IP Address** | **OS/Model** | **Port** |
| --- | --- | --- | --- | --- |
| 1 | www.sac.or.th | 10.1.1.10 | - | TCP: 22, 80, 443, 8008, 8010 |
| 2 | lib.sac.or.th | 10.1.1.121 | - | TCP: 21, 80, 443, 3389, 8008, 8010, 8081 |

## **5.2 Executive summary**

The purpose of this activity is to find the vulnerability on the target web application.

### **5.2.1 Summary Vulnerability by Severity**

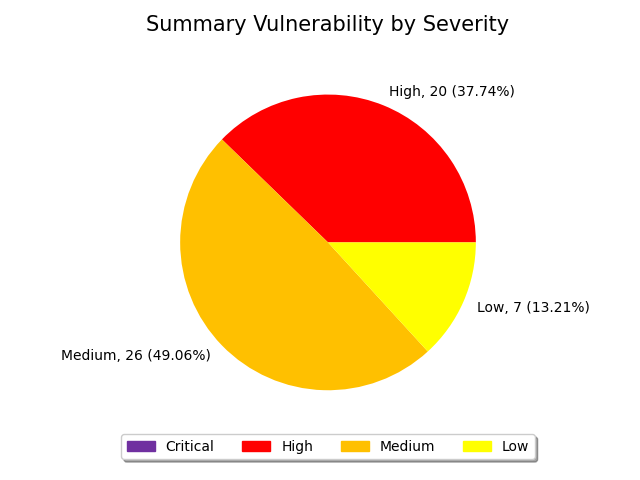


Figure 6: Summary by Severity of Web Application Vulnerability Assessment

### **5.2.2 Vulnerability by Target**

| **No.** | **Domain/Server Name** | **IP Address** | **Critical** | **High** | **Medium** | **Low** | **Total** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | https://localhost:8888 | 127.0.0.1 | 0 | 20 | 26 | 7 | 53 |
| **Total** | | | 0 | 20 | 26 | 7 | 53 |

## **5.3 Web Application Vulnerability Detail**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 1 | **Finding** | SQL injection |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/sqli\_blind/ [id parameter] https://localhost:8888/dvwa/vulnerabilities/brute/ [username parameter] https://localhost:8888/dvwa/vulnerabilities/sqli/ [id parameter] | | |
| **Detail** | SQL injection vulnerabilities arise when user-controllable data is incorporated into database SQL queries in an unsafe manner. An attacker can supply crafted input to break out of the data context in which their input appears and interfere with the structure of the surrounding query. A wide range of damaging attacks can often be delivered via SQL injection, including reading or modifying critical application data, interfering with application logic, escalating privileges within the database and taking control of the database server. | | |
| **Solution** | The most effective way to prevent SQL injection attacks is to use parameterized queries (also known as prepared statements) for all database access. This method uses two steps to incorporate potentially tainted data into SQL queries: first, the application specifies the structure of the query, leaving placeholders for each item of user input; second, the application specifies the contents of each placeholder. Because the structure of the query has already been defined in the first step, it is not possible for malformed data in the second step to interfere with the query structure. You should review the documentation for your database and application platform to determine the appropriate APIs which you can use to perform parameterized queries. It is strongly recommended that you parameterize every variable data item that is incorporated into database queries, even if it is not obviously tainted, to prevent oversights occurring and avoid vulnerabilities being introduced by changes elsewhere within the code base of the application. You should be aware that some commonly employed and recommended mitigations for SQL injection vulnerabilities are not always effective:    One common defense is to double up any single quotation marks appearing within user input before incorporating that input into a SQL query. This defense is designed to prevent malformed data from terminating the string into which it is inserted. However, if the data being incorporated into queries is numeric, then the defense may fail, because numeric data may not be encapsulated within quotes, in which case only a space is required to break out of the data context and interfere with the query. Further, in second-order SQL injection attacks, data that has been safely escaped when initially inserted into the database is subsequently read from the database and then passed back to it again. Quotation marks that have been doubled up initially will return to their original form when the data is reused, allowing the defense to be bypassed.  Another often cited defense is to use stored procedures for database access. While stored procedures can provide security benefits, they are not guaranteed to prevent SQL injection attacks. The same kinds of vulnerabilities that arise within standard dynamic SQL queries can arise if any SQL is dynamically constructed within stored procedures. Further, even if the procedure is sound, SQL injection can arise if the procedure is invoked in an unsafe manner using user-controllable data. | | |
| **Remark** | https://portswigger.net/web-security/sql-injection https://support.portswigger.net/customer/portal/articles/1965677-using-burp-to-test-for-injection-flaws https://portswigger.net/web-security/sql-injection/cheat-sheet | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 2 | **Finding** | OS command injection |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/exec/ [ip parameter] | | |
| **Detail** | Operating system command injection vulnerabilities arise when an application incorporates user-controllable data into a command that is processed by a shell command interpreter. If the user data is not strictly validated, an attacker can use shell metacharacters to modify the command that is executed, and inject arbitrary further commands that will be executed by the server. OS command injection vulnerabilities are usually very serious and may lead to compromise of the server hosting the application, or of the application's own data and functionality. It may also be possible to use the server as a platform for attacks against other systems. The exact potential for exploitation depends upon the security context in which the command is executed, and the privileges that this context has regarding sensitive resources on the server. | | |
| **Solution** | If possible, applications should avoid incorporating user-controllable data into operating system commands. In almost every situation, there are safer alternative methods of performing server-level tasks, which cannot be manipulated to perform additional commands than the one intended. If it is considered unavoidable to incorporate user-supplied data into operating system commands, the following two layers of defense should be used to prevent attacks:    The user data should be strictly validated. Ideally, a whitelist of specific accepted values should be used. Otherwise, only short alphanumeric strings should be accepted. Input containing any other data, including any conceivable shell metacharacter or whitespace, should be rejected.  The application should use command APIs that launch a specific process via its name and command-line parameters, rather than passing a command string to a shell interpreter that supports command chaining and redirection. For example, the Java API Runtime.exec and the ASP.NET API Process.Start do not support shell metacharacters. This defense can mitigate the impact of an attack even in the event that an attacker circumvents the input validation defenses. | | |
| **Remark** | https://portswigger.net/web-security/os-command-injection | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 3 | **Finding** | Cross-site scripting (reflected) |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/view\_source.php [security parameter] https://localhost:8888/dvwa/vulnerabilities/view\_source.php [id parameter] https://localhost:8888/dvwa/vulnerabilities/csrf/test\_credentials.php [username parameter] https://localhost:8888/dvwa/vulnerabilities/csp/ [include parameter] https://localhost:8888/dvwa/vulnerabilities/xss\_r/ [name parameter] | | |
| **Detail** | Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application. The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes. Users can be induced to issue the attacker's crafted request in various ways. For example, the attacker can send a victim a link containing a malicious URL in an email or instant message. They can submit the link to popular web sites that allow content authoring, for example in blog comments. And they can create an innocuous looking web site that causes anyone viewing it to make arbitrary cross-domain requests to the vulnerable application (using either the GET or the POST method). The security impact of cross-site scripting vulnerabilities is dependent upon the nature of the vulnerable application, the kinds of data and functionality that it contains, and the other applications that belong to the same domain and organization. If the application is used only to display non-sensitive public content, with no authentication or access control functionality, then a cross-site scripting flaw may be considered low risk. However, if the same application resides on a domain that can access cookies for other more security-critical applications, then the vulnerability could be used to attack those other applications, and so may be considered high risk. Similarly, if the organization that owns the application is a likely target for phishing attacks, then the vulnerability could be leveraged to lend credibility to such attacks, by injecting Trojan functionality into the vulnerable application and exploiting users' trust in the organization in order to capture credentials for other applications that it owns. In many kinds of application, such as those providing online banking functionality, cross-site scripting should always be considered high risk. | | |
| **Solution** | In most situations where user-controllable data is copied into application responses, cross-site scripting   attacks can be prevented using two layers of defenses:   Input should be validated as strictly as possible on arrival, given the kind of content that  it is expected to contain. For example, personal names should consist of alphabetical  and a small range of typographical characters, and be relatively short; a year of birth  should consist of exactly four numerals; email addresses should match a well-defined  regular expression. Input which fails the validation should be rejected, not sanitized. User input should be HTML-encoded at any point where it is copied into  application responses. All HTML metacharacters, including < > " ' and =, should be  replaced with the corresponding HTML entities (&lt; &gt; etc). In cases where the application's functionality allows users to author content using   a restricted subset of HTML tags and attributes (for example, blog comments which   allow limited formatting and linking), it is necessary to parse the supplied HTML to   validate that it does not use any dangerous syntax; this is a non-trivial task. | | |
| **Remark** | https://portswigger.net/web-security/cross-site-scripting https://portswigger.net/web-security/cross-site-scripting/reflected https://support.portswigger.net/customer/portal/articles/1965737-Methodology\_XSS.html | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 4 | **Finding** | External service interaction (HTTP) |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/fi/ [page parameter] | | |
| **Detail** | External service interaction arises when it is possible to induce an application to interact with an arbitrary external service, such as a web or mail server. The ability to trigger arbitrary external service interactions does not constitute a vulnerability in its own right, and in some cases might even be the intended behavior of the application. However, in many cases, it can indicate a vulnerability with serious consequences. The ability to send requests to other systems can allow the vulnerable server to be used as an attack proxy.  By submitting suitable payloads, an attacker can cause the application server to attack other systems that it can interact with.   This may include public third-party systems, internal systems within the same organization, or services available on the local loopback adapter of the application server itself.   Depending on the network architecture, this may expose highly vulnerable internal services that are not otherwise accessible to external attackers. | | |
| **Solution** | You should review the purpose and intended use of the relevant application functionality,   and determine whether the ability to trigger arbitrary external service interactions is intended behavior.   If so, you should be aware of the types of attacks that can be performed via this behavior and take appropriate measures.   These measures might include blocking network access from the application server to other internal systems, and hardening the application server itself to remove any services available on the local loopback adapter. If the ability to trigger arbitrary external service interactions is not intended behavior, then you should implement a whitelist of permitted services and hosts, and block any interactions that do not appear on this whitelist.  Out-of-Band Application Security Testing (OAST) is highly effective at uncovering high-risk features, to the point where finding the root cause of an interaction can be quite challenging. To find the source of an external service interaction, try to identify whether it is triggered by specific application functionality, or occurs indiscriminately on all requests. If it occurs on all endpoints, a front-end CDN or application firewall may be responsible, or a back-end analytics system parsing server logs. In some cases, interactions may originate from third-party systems; for example, a HTTP request may trigger a poisoned email which passes through a link-scanner on its way to the recipient. | | |
| **Remark** | https://portswigger.net/blog/introducing-burp-collaborator https://portswigger.net/burp/application-security-testing/oast https://portswigger.net/research/cracking-the-lens-targeting-https-hidden-attack-surface | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 5 | **Finding** | External service interaction (DNS) |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/fi/ [page parameter] | | |
| **Detail** | External service interaction arises when it is possible to induce an application to interact with an arbitrary external service, such as a web or mail server. The ability to trigger arbitrary external service interactions does not constitute a vulnerability in its own right, and in some cases might even be the intended behavior of the application. However, in many cases, it can indicate a vulnerability with serious consequences. In cases where DNS-based interactions can be triggered, it is normally possible to trigger interactions using other service types, and these are reported as separate issues.   If a payload that specifies a particular service type (e.g. a URL) triggers only a DNS-based interaction,   then this strongly indicates that the application attempted to connect using that other service,   but was prevented from doing so by egress filters in place at the network layer. The ability to send requests to other systems can allow the vulnerable server to be used as an attack proxy.  By submitting suitable payloads, an attacker can cause the application server to attack other systems that it can interact with.   This may include public third-party systems, internal systems within the same organization, or services available on the local loopback adapter of the application server itself.   Depending on the network architecture, this may expose highly vulnerable internal services that are not otherwise accessible to external attackers. | | |
| **Solution** | You should review the purpose and intended use of the relevant application functionality,   and determine whether the ability to trigger arbitrary external service interactions is intended behavior.   If so, you should be aware of the types of attacks that can be performed via this behavior and take appropriate measures.   These measures might include blocking network access from the application server to other internal systems, and hardening the application server itself to remove any services available on the local loopback adapter. If the ability to trigger arbitrary external service interactions is not intended behavior, then you should implement a whitelist of permitted services and hosts, and block any interactions that do not appear on this whitelist.  Out-of-Band Application Security Testing (OAST) is highly effective at uncovering high-risk features, to the point where finding the root cause of an interaction can be quite challenging. To find the source of an external service interaction, try to identify whether it is triggered by specific application functionality, or occurs indiscriminately on all requests. If it occurs on all endpoints, a front-end CDN or application firewall may be responsible, or a back-end analytics system parsing server logs. In some cases, interactions may originate from third-party systems; for example, a HTTP request may trigger a poisoned email which passes through a link-scanner on its way to the recipient. | | |
| **Remark** | https://portswigger.net/blog/introducing-burp-collaborator https://portswigger.net/burp/application-security-testing/oast https://portswigger.net/research/cracking-the-lens-targeting-https-hidden-attack-surface | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 6 | **Finding** | Cross-site scripting (DOM-based) |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/xss\_d/ | | |
| **Detail** | DOM-based vulnerabilities arise when a client-side script reads data from a controllable part of the DOM (for example, the URL) and processes this data in an unsafe way. DOM-based cross-site scripting arises when a script writes controllable data into the HTML document in an unsafe way. An attacker may be able to use the vulnerability to construct a URL that, if visited by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application. The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes. Users can be induced to visit the attacker's crafted URL in various ways, similar to the usual attack delivery vectors for reflected cross-site scripting vulnerabilities.  Burp Suite automatically identifies this issue using static code analysis, which may lead to false positives that are not actually exploitable. The relevant code and execution paths should be reviewed to determine whether this vulnerability is indeed present, or whether mitigations are in place that would prevent exploitation. | | |
| **Solution** | The most effective way to avoid DOM-based cross-site scripting vulnerabilities is not to dynamically write data from any untrusted source into the HTML document. If the desired functionality of the application means that this behavior is unavoidable, then defenses must be implemented within the client-side code to prevent malicious data from introducing script code into the document. In many cases, the relevant data can be validated on a whitelist basis, to allow only content that is known to be safe. In other cases, it will be necessary to sanitize or encode the data. This can be a complex task, and depending on the context that the data is to be inserted may need to involve a combination of JavaScript escaping, HTML encoding, and URL encoding, in the appropriate sequence. | | |
| **Remark** | https://portswigger.net/web-security/cross-site-scripting https://portswigger.net/web-security/cross-site-scripting/dom-based | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 7 | **Finding** | File path traversal |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/fi/ [page parameter] | | |
| **Detail** | File path traversal vulnerabilities arise when user-controllable data is used within a filesystem operation in an unsafe manner. Typically, a user-supplied filename is appended to a directory prefix in order to read or write the contents of a file. If vulnerable, an attacker can supply path traversal sequences (using dot-dot-slash characters) to break out of the intended directory and read or write files elsewhere on the filesystem. This is typically a very serious vulnerability, enabling an attacker to access sensitive files containing configuration data, passwords, database records, log data, source code, and program scripts and binaries. | | |
| **Solution** | Ideally, application functionality should be designed in such a way that user-controllable data does not need to be passed to filesystem operations. This can normally be achieved by referencing known files via an index number rather than their name, and using application-generated filenames to save user-supplied file content. If it is considered unavoidable to pass user-controllable data to a filesystem operation, three layers of defense can be employed to prevent path traversal attacks:  User-controllable data should be strictly validated before being passed to any filesystem operation. In particular, input containing dot-dot sequences should be blocked.After validating user input, the application can use a suitable filesystem API to verify that the file to be accessed is actually located within the base directory used by the application. In Java, this can be achieved by instantiating a java.io.File object using the user-supplied filename and then calling the getCanonicalPath method on this object. If the string returned by this method does not begin with the name of the start directory, then the user has somehow bypassed the application's input filters, and the request should be rejected. In ASP.NET, the same check can be performed by passing the user-supplied filename to the System.Io.Path.GetFullPath method and checking the returned string in the same way as described for Java.The directory used to store files that are accessed using user-controllable data can be located on a separate logical volume to other sensitive application and operating system files, so that these cannot be reached via path traversal attacks. In Unix-based systems, this can be achieved using a chrooted filesystem; on Windows, this can be achieved by mounting the base directory as a new logical drive and using the associated drive letter to access its contents. | | |
| **Remark** | https://portswigger.net/web-security/file-path-traversal | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 8 | **Finding** | Out-of-band resource load (HTTP) |
| **Severity** | **High** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/fi/ [page parameter] | | |
| **Detail** | Out-of-band resource load arises when it is possible to induce an application to fetch content from an arbitrary external location, and incorporate that content into the application's own response(s). The ability to trigger arbitrary out-of-band resource load does not constitute a vulnerability in its own right, and in some cases might even be the intended behavior of the application.  However, in many cases, it can indicate a vulnerability with serious consequences. The ability to request and retrieve web content from other systems can allow the application server to be used as a two-way attack proxy.   By submitting suitable payloads, an attacker can cause the application server to attack, or retrieve content from, other systems that it can interact with.   This may include public third-party systems, internal systems within the same organization, or services available on the local loopback adapter of the application server itself.   Depending on the network architecture, this may expose highly vulnerable internal services that are not otherwise accessible to external attackers. Additionally, the application's processing of web content that is retrieved from arbitrary URLs exposes some important and non-conventional attack surface.   An attacker can deploy a web server that returns malicious content, and then induce the application to retrieve and process that content.   This processing might give rise to the types of input-based vulnerabilities that are normally found when unexpected input is submitted directly in requests to the application.   The out-of-band attack surface that the application exposes should be thoroughly tested for these types of vulnerabilities. | | |
| **Solution** | You should review the purpose and intended use of the relevant application functionality, and determine whether the ability to trigger arbitrary out-of-band resource load is intended behavior.   If so, you should be aware of the types of attacks that can be performed via this behavior and take appropriate measures.   These measures might include blocking network access from the application server to other internal systems, and hardening the application server itself to remove any services available on the local loopback adapter.  You should also ensure that content retrieved from other systems is processed in a safe manner, with the usual precautions that are applicable when processing input from direct incoming web requests. If the ability to trigger arbitrary out-of-band resource load is not intended behavior, then you should implement a whitelist of permitted URLs, and block requests to URLs that do not appear on this whitelist. | | |
| **Remark** | https://portswigger.net/blog/introducing-burp-collaborator https://portswigger.net/burp/application-security-testing/oast https://portswigger.net/research/cracking-the-lens-targeting-https-hidden-attack-surface | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 9 | **Finding** | Cross-site request forgery |
| **Severity** | **Medium** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/captcha/ https://localhost:8888/dvwa/vulnerabilities/upload/ https://localhost:8888/dvwa/vulnerabilities/javascript/ https://localhost:8888/dvwa/vulnerabilities/csp/ https://localhost:8888/dvwa/vulnerabilities/weak\_id/ | | |
| **Detail** | Cross-site request forgery (CSRF) vulnerabilities may arise when applications rely solely on HTTP cookies to identify the user that has issued a particular request. Because browsers automatically add cookies to requests regardless of their origin, it may be possible for an attacker to create a malicious web site that forges a cross-domain request to the vulnerable application. For a request to be vulnerable to CSRF, the following conditions must hold:  The request can be issued cross-domain, for example using an HTML form. If the request contains non-standard headers or body content, then it may only be issuable from a page that originated on the same domain. The application relies solely on HTTP cookies or Basic Authentication to identify the user that issued the request. If the application places session-related tokens elsewhere within the request, then it may not be vulnerable. The request performs some privileged action within the application, which modifies the application's state based on the identity of the issuing user.The attacker can determine all the parameters required to construct a request that performs the action. If the request contains any values that the attacker cannot determine or predict, then it is not vulnerable. | | |
| **Solution** | The most effective way to protect against CSRF vulnerabilities is to include within relevant requests an additional token that is not transmitted in a cookie: for example, a parameter in a hidden form field. This additional token should contain sufficient entropy, and be generated using a cryptographic random number generator, such that it is not feasible for an attacker to determine or predict the value of any token that was issued to another user. The token should be associated with the user's session, and the application should validate that the correct token is received before performing any action resulting from the request. An alternative approach, which may be easier to implement, is to validate that Host and Referer headers in relevant requests are both present and contain the same domain name. However, this approach is somewhat less robust: historically, quirks in browsers and plugins have often enabled attackers to forge cross-domain requests that manipulate these headers to bypass such defenses. | | |
| **Remark** | https://portswigger.net/web-security/csrf https://support.portswigger.net/customer/portal/articles/1965674-using-burp-to-test-for-cross-site-request-forgery-csrf- https://media.blackhat.com/eu-13/briefings/Lundeen/bh-eu-13-deputies-still-confused-lundeen-wp.pdf | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 10 | **Finding** | Cross-site scripting (reflected) |
| **Severity** | **Medium** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/csrf/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/sqli/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/upload/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/javascript/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/sqli\_blind/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/xss\_s/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/csp/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/xss\_r/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/captcha/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/brute/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/fi/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/weak\_id/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/exec/ [security cookie] https://localhost:8888/dvwa/vulnerabilities/xss\_d/ [security cookie] | | |
| **Detail** | Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application. The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes. Users can be induced to issue the attacker's crafted request in various ways. For example, the attacker can send a victim a link containing a malicious URL in an email or instant message. They can submit the link to popular web sites that allow content authoring, for example in blog comments. And they can create an innocuous looking web site that causes anyone viewing it to make arbitrary cross-domain requests to the vulnerable application (using either the GET or the POST method). The security impact of cross-site scripting vulnerabilities is dependent upon the nature of the vulnerable application, the kinds of data and functionality that it contains, and the other applications that belong to the same domain and organization. If the application is used only to display non-sensitive public content, with no authentication or access control functionality, then a cross-site scripting flaw may be considered low risk. However, if the same application resides on a domain that can access cookies for other more security-critical applications, then the vulnerability could be used to attack those other applications, and so may be considered high risk. Similarly, if the organization that owns the application is a likely target for phishing attacks, then the vulnerability could be leveraged to lend credibility to such attacks, by injecting Trojan functionality into the vulnerable application and exploiting users' trust in the organization in order to capture credentials for other applications that it owns. In many kinds of application, such as those providing online banking functionality, cross-site scripting should always be considered high risk. | | |
| **Solution** | In most situations where user-controllable data is copied into application responses, cross-site scripting   attacks can be prevented using two layers of defenses:   Input should be validated as strictly as possible on arrival, given the kind of content that  it is expected to contain. For example, personal names should consist of alphabetical  and a small range of typographical characters, and be relatively short; a year of birth  should consist of exactly four numerals; email addresses should match a well-defined  regular expression. Input which fails the validation should be rejected, not sanitized. User input should be HTML-encoded at any point where it is copied into  application responses. All HTML metacharacters, including < > " ' and =, should be  replaced with the corresponding HTML entities (&lt; &gt; etc). In cases where the application's functionality allows users to author content using   a restricted subset of HTML tags and attributes (for example, blog comments which   allow limited formatting and linking), it is necessary to parse the supplied HTML to   validate that it does not use any dangerous syntax; this is a non-trivial task. | | |
| **Remark** | https://portswigger.net/web-security/cross-site-scripting https://portswigger.net/web-security/cross-site-scripting/reflected https://support.portswigger.net/customer/portal/articles/1965737-Methodology\_XSS.html | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 11 | **Finding** | TLS cookie without secure flag set |
| **Severity** | **Medium** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/security.php https://localhost:8888/dvwa/ | | |
| **Detail** | If the secure flag is set on a cookie, then browsers will not submit the cookie in any requests that use an unencrypted HTTP connection, thereby preventing the cookie from being trivially intercepted by an attacker monitoring network traffic. If the secure flag is not set, then the cookie will be transmitted in clear-text if the user visits any HTTP URLs within the cookie's scope. An attacker may be able to induce this event by feeding a user suitable links, either directly or via another web site. Even if the domain that issued the cookie does not host any content that is accessed over HTTP, an attacker may be able to use links of the form http://example.com:443/ to perform the same attack. To exploit this vulnerability, an attacker must be suitably positioned to eavesdrop on the victim's network traffic. This scenario typically occurs when a client communicates with the server over an insecure connection such as public Wi-Fi, or a corporate or home network that is shared with a compromised computer. Common defenses such as switched networks are not sufficient to prevent this. An attacker situated in the user's ISP or the application's hosting infrastructure could also perform this attack. Note that an advanced adversary could potentially target any connection made over the Internet's core infrastructure. | | |
| **Solution** | The secure flag should be set on all cookies that are used for transmitting sensitive data when accessing content over HTTPS. If cookies are used to transmit session tokens, then areas of the application that are accessed over HTTPS should employ their own session handling mechanism, and the session tokens used should never be transmitted over unencrypted communications. | | |
| **Remark** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 12 | **Finding** | TLS certificate |
| **Severity** | **Medium** | **Port** | 8888 |
| **Target** | https://localhost:8888/ | | |
| **Detail** | TLS (or SSL) helps to protect the confidentiality and integrity of information in transit between the browser and server, and to provide authentication of the server's identity. To serve this purpose, the server must present an TLS certificate that is valid for the server's hostname, is issued by a trusted authority and is valid for the current date. If any one of these requirements is not met, TLS connections to the server will not provide the full protection for which TLS is designed. It should be noted that various attacks exist against TLS in general, and in the context of HTTPS web connections in particular. It may be possible for a determined and suitably-positioned attacker to compromise TLS connections without user detection even when a valid TLS certificate is used. | | |
| **Solution** |  | | |
| **Remark** | https://wiki.mozilla.org/Security/Server\_Side\_TLS | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 13 | **Finding** | Cross-site scripting (reflected) |
| **Severity** | **Low** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/fi/ [Referer HTTP header] https://localhost:8888/dvwa/vulnerabilities/fi/ [User-Agent HTTP header] | | |
| **Detail** | Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application. The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes. Users can be induced to issue the attacker's crafted request in various ways. For example, the attacker can send a victim a link containing a malicious URL in an email or instant message. They can submit the link to popular web sites that allow content authoring, for example in blog comments. And they can create an innocuous looking web site that causes anyone viewing it to make arbitrary cross-domain requests to the vulnerable application (using either the GET or the POST method). The security impact of cross-site scripting vulnerabilities is dependent upon the nature of the vulnerable application, the kinds of data and functionality that it contains, and the other applications that belong to the same domain and organization. If the application is used only to display non-sensitive public content, with no authentication or access control functionality, then a cross-site scripting flaw may be considered low risk. However, if the same application resides on a domain that can access cookies for other more security-critical applications, then the vulnerability could be used to attack those other applications, and so may be considered high risk. Similarly, if the organization that owns the application is a likely target for phishing attacks, then the vulnerability could be leveraged to lend credibility to such attacks, by injecting Trojan functionality into the vulnerable application and exploiting users' trust in the organization in order to capture credentials for other applications that it owns. In many kinds of application, such as those providing online banking functionality, cross-site scripting should always be considered high risk. | | |
| **Solution** | In most situations where user-controllable data is copied into application responses, cross-site scripting   attacks can be prevented using two layers of defenses:   Input should be validated as strictly as possible on arrival, given the kind of content that  it is expected to contain. For example, personal names should consist of alphabetical  and a small range of typographical characters, and be relatively short; a year of birth  should consist of exactly four numerals; email addresses should match a well-defined  regular expression. Input which fails the validation should be rejected, not sanitized. User input should be HTML-encoded at any point where it is copied into  application responses. All HTML metacharacters, including < > " ' and =, should be  replaced with the corresponding HTML entities (&lt; &gt; etc). In cases where the application's functionality allows users to author content using   a restricted subset of HTML tags and attributes (for example, blog comments which   allow limited formatting and linking), it is necessary to parse the supplied HTML to   validate that it does not use any dangerous syntax; this is a non-trivial task. | | |
| **Remark** | https://portswigger.net/web-security/cross-site-scripting https://portswigger.net/web-security/cross-site-scripting/reflected https://support.portswigger.net/customer/portal/articles/1965737-Methodology\_XSS.html | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 14 | **Finding** | Cookie without HttpOnly flag set |
| **Severity** | **Low** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/security.php https://localhost:8888/dvwa/ | | |
| **Detail** | If the HttpOnly attribute is set on a cookie, then the cookie's value cannot be read or set by client-side JavaScript. This measure makes certain client-side attacks, such as cross-site scripting, slightly harder to exploit by preventing them from trivially capturing the cookie's value via an injected script. | | |
| **Solution** | There is usually no good reason not to set the HttpOnly flag on all cookies. Unless you specifically require legitimate client-side scripts within your application to read or set a cookie's value, you should set the HttpOnly flag by including this attribute within the relevant Set-cookie directive. You should be aware that the restrictions imposed by the HttpOnly flag can potentially be circumvented in some circumstances, and that numerous other serious attacks can be delivered by client-side script injection, aside from simple cookie stealing. | | |
| **Remark** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 15 | **Finding** | Password submitted using GET method |
| **Severity** | **Low** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa/vulnerabilities/brute/ https://localhost:8888/dvwa/vulnerabilities/csrf/ | | |
| **Detail** | Some applications use the GET method to submit passwords, which are transmitted within the query string of the requested URL. Sensitive information within URLs may be logged in various locations, including the user's browser, the web server, and any forward or reverse proxy servers between the two endpoints. URLs may also be displayed on-screen, bookmarked or emailed around by users. They may be disclosed to third parties via the Referer header when any off-site links are followed. Placing passwords into the URL increases the risk that they will be captured by an attacker. Vulnerabilities that result in the disclosure of users' passwords can result in compromises that are extremely difficult to investigate due to obscured audit trails. Even if the application itself only handles non-sensitive information, exposing passwords puts users who have re-used their password elsewhere at risk. | | |
| **Solution** | All forms submitting passwords should use the POST method. To achieve this, applications should specify the method attribute of the FORM tag as method="POST". It may also be necessary to modify the corresponding server-side form handler to ensure that submitted passwords are properly retrieved from the message body, rather than the URL. | | |
| **Remark** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID.** | 16 | **Finding** | Strict transport security not enforced |
| **Severity** | **Low** | **Port** | 8888 |
| **Target** | https://localhost:8888/dvwa | | |
| **Detail** | The application fails to prevent users from connecting to it over unencrypted connections. An attacker able to modify a legitimate user's network traffic could bypass the application's use of SSL/TLS encryption, and use the application as a platform for attacks against its users. This attack is performed by rewriting HTTPS links as HTTP, so that if a targeted user follows a link to the site from an HTTP page, their browser never attempts to use an encrypted connection. The sslstrip tool automates this process.   To exploit this vulnerability, an attacker must be suitably positioned to intercept and modify the victim's network traffic.This scenario typically occurs when a client communicates with the server over an insecure connection such as public Wi-Fi, or a corporate or home network that is shared with a compromised computer. Common defenses such as switched networks are not sufficient to prevent this. An attacker situated in the user's ISP or the application's hosting infrastructure could also perform this attack. Note that an advanced adversary could potentially target any connection made over the Internet's core infrastructure. | | |
| **Solution** | The application should instruct web browsers to only access the application using HTTPS. To do this, enable HTTP Strict Transport Security (HSTS) by adding a response header with the name 'Strict-Transport-Security' and the value 'max-age=expireTime', where expireTime is the time in seconds that browsers should remember that the site should only be accessed using HTTPS. Consider adding the 'includeSubDomains' flag if appropriate. Note that because HSTS is a "trust on first use" (TOFU) protocol, a user who has never accessed the application will never have seen the HSTS header, and will therefore still be vulnerable to SSL stripping attacks. To mitigate this risk, you can optionally add the 'preload' flag to the HSTS header, and submit the domain for review by browser vendors. | | |
| **Remark** | https://developer.mozilla.org/en-US/docs/Web/Security/HTTP\_strict\_transport\_security https://github.com/moxie0/sslstrip https://hstspreload.appspot.com/ | | |



# Port Discovery

| **Port** | **Protocol** | **Service** |
| --- | --- | --- |
| 21 | tcp | ftp |
| 22 | tcp | ssh |
| 80 | tcp | http |
| 443 | tcp | https |
| 3389 | tcp | ms-wbt-server |
| 8008 | tcp | http |
| 8010 | tcp | xmpp |
| 8081 | tcp | blackice-icecap |

# Appendix

## **7.1 About Nessus**

Nessus is a proprietary vulnerability scanner developed by Tenable, Inc. Nessus is trusted by more than 30,000 organizations worldwide as one of the most widely deployed security technologies on the planet - and the gold standard for vulnerability assessment.

Reference: https://www.tenable.com/products/nessus

### **7.1.1 Nessus vulnerabilities**

As information about new vulnerabilities are discovered and released into the public domain, Tenable, Inc. research staff designs programs to enable Nessus to detect them. These programs are named plugins, and are written in the Nessus proprietary scripting language, called Nessus Attack Scripting Language (NASL). Plugins contain vulnerability information, a generic set of remediation actions, and the algorithm to test for the presence of the security issue.

Reference: https://www.tenable.com/plugins

### **7.1.2 Nessus risk score**

There are four risk levels in this document: Critical, High, Medium, and Low. There are methods for determining the risk level. Based on the Common Vulnerability Scoring System (CVSS), a standard for assessing the severity of vulnerabilities in computer systems. Regarded by the NIAC (National Infrastructure Advisory Council), expert assessments are measured in a range of 0 – 10

| **Severity** | **Description** | **Score** |
| --- | --- | --- |
| Critical | Vulnerabilities that score in the critical range usually have most of the following characteristics:   * Exploitation of the vulnerability likely results in root-level compromise of servers or infrastructure devices. * Exploitation is usually straightforward, in the sense that the attacker does not need any special authentication credentials or knowledge about individual victims, and does not need to persuade a target user, for example via social engineering, into performing any special functions.   For critical vulnerabilities, is advised that you patch or upgrade as soon as possible, unless you have other mitigating measures in place. For example, a mitigating factor could be if your installation is not accessible from the Internet. | 9.0 – 10.0 |
| High | Vulnerabilities that score in the high range usually have some of the following characteristics:   * The vulnerability is difficult to exploit. * Exploitation could result in elevated privileges. * Exploitation could result in a significant data loss or downtime. | 7.0 – 8.9 |
| Medium | Vulnerabilities that score in the medium range usually have some of the following characteristics:   * Vulnerabilities that require the attacker to manipulate individual victims via social engineering tactics. * Denial of service vulnerabilities that are difficult to set up. * Exploits that require an attacker to reside on the same local network as the victim. * Vulnerabilities where exploitation provides only very limited access. * Vulnerabilities that require user privileges for successful exploitation. | 4.0 – 6.9 |
| Low | Vulnerabilities in the low range typically have very little impact on an organization's business. Exploitation of such vulnerabilities usually requires local or physical system access. | 0.1 – 3.9 |

## **7.2 About Burp Suite's web vulnerability scanner**

The web vulnerability scanner behind Burp Suite's popularity has more to it than most. Burp Scanner uses PortSwigger's world-leading research to help its users find a wide range of vulnerabilities in web applications, automatically. Sitting at the core of both Burp Suite Enterprise Edition and Burp Suite Professional, Burp Scanner is the weapon of choice for over 60,000 users across more than 15,000 organizations.

Reference: https://portswigger.net/burp/vulnerability-scanner

### **7.2.1 Burp Suite's web vulnerability scanner risk score**

The level of severity for an issue that was found by a scan. The higher the severity level, the larger the impact is likely to be if an attacker is able to exploit this vulnerability. Note that the severity level is only a rough approximation based on a typical website. You should use your knowledge of the purpose and context of the associated functionality to determine how serious each issue is in your individual case.

Reference: https://portswigger.net/burp/extensibility/enterprise/graphql-api/severity.html

| **Severity** | **Description** |
| --- | --- |
| High | An attacker can **fully** compromise the confidentiality, integrity, or availability, of a target system without specialized access, user interaction or circumstances that are beyond the attacker’s control. Very likely to allow lateral movement and escalation of attack to other systems on the internal network of the vulnerable application. |
| Medium | An attacker can **partially** compromise the confidentiality, integrity, or availability, of a target system. Specialized access, user interaction, or circumstances that are beyond the attacker’s control may be required for an attack to succeed. Very likely to be used in conjunction with other vulnerabilities to escalate an attack. |
| Low | An attacker can **limitedly** compromise the confidentiality, integrity, or availability, of a target system. Specialized access, user interaction, or circumstances that are beyond the attacker’s control is required for an attack to succeed. Needs to be used in conjunction with other vulnerabilities to escalate an attack. |