**TEST**

**Dynamic Application Security Test (DAST)**

**10/10/2022**

Contents

[1. Executive Summary 3](#_Toc99805179)

[1.1 Overview 3](#_Toc99805180)

[1.1.1 Threat Overview 3](#_Toc99805181)

[1.2 Project Objectives 3](#_Toc99805182)

[1.3 Assessment Details 3](#_Toc99805183)

[2. Findings 5](#_Toc99805184)

[2.1 Severity Findings 5](#_Toc99805185)

[2.1.1 5](#_Toc99805186)

[3. Appendix 6](#_Toc99805187)

[3.1 Remediation Requirements 6](#_Toc99805188)

**Disclaimer**

<Replace me with Confidentiality policy>

# Executive Summary

## Overview

Full dynamic application security testing was conducted from 10/10/2022 through 10/10/2022.

Automated dynamic application security testing using Burp Suite was performed on the API due to Application Security Standard Requirements.

**Disclaimer:**

<Please fill if available>.

### Threat Overview

Overall, identified the following number of security issues:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Number of Observations by Threat Level** | | | | | | |
|  | **Total** | **Critical** | **High** | **Medium** | **Low** | **Informational** |
| **Open** | **0** | 0 | 0 | 0 | 0 | 0 |
| **Closed** | 0 | 0 | 0 | 0 | 0 | 0 |

| **Issue Name** | **Severity** | **Finding ID** |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Project Objectives

The following IP/URLs were tested for security vulnerabilities:

* http://s96141-102131-t7y.croto.hack.me : 74.50.111.247

## Assessment Details

|  |  |
| --- | --- |
| **Application Name:** | **TEST\_API** |
| **MAL Code:** | **TEST** |
| **Line of Business:** | **PROD** |
| **Application Owner:** | **Rkannan** |
| **Assessment Name:** | **Dynamic Scan for Web Application** |
| **Intake Request:** | **API1001** |
| **Requester Name:** | **CSO** |
| **Assessor Name:** | **Tester** |
| **Assessment Completed Date:** | **10/10/2022** |
| **Technical Contact:** | **John** |
| **Project Identifier / Ref:** | **010** |

# Findings

## Critical Severity Findings

|  |
| --- |
|  |

## High Severity Findings

|  |  |
| --- | --- |
| Cleartext submission of password | |
| Severity: High | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/index.php | GET |
| Vulnerability Details | |
| The page contains a form with the following action URL, which is submitted over clear-text HTTP:http://s96141-102131-t7y.croto.hack.me/index.phpThe form contains the following password field:password | |
| Some applications transmit passwords over unencrypted connections, making them vulnerable to interception. 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. 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. | |
| Recommendation | |
| Applications should use transport-level encryption (SSL or TLS) to protect all sensitive communications passing between the client and the server. Communications that should be protected include the login mechanism and related functionality, and any functions where sensitive data can be accessed or privileged actions can be performed. These areas should employ their own session handling mechanism, and the session tokens used should never be transmitted over unencrypted communications. If HTTP cookies are used for transmitting session tokens, then the secure flag should be set to prevent transmission over clear-text HTTP. | |
| References | |
|  | |
| SQL injection | |
| Severity: High | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/index.php | POST |
| Vulnerability Details | |
| The username parameter appears to be vulnerable to SQL injection attacks. A single quote was submitted in the username parameter, and a general error message was returned. Two single quotes were then submitted and the error message disappeared. You should review the contents of the error message, and the application's handling of other input, to confirm whether a vulnerability is present. | |
| 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. | |
| Recommendation | |
| 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. | |
| References | |
|  | |
|  | |
|  | |
|  | |
| Cross-site scripting (reflected) | |
| Severity: High | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/index.php | GET |
| Vulnerability Details | |
| The value of the username request parameter is copied into the HTML document as plain text between tags. The payload edlil/lt;script/gt;alert(1)/lt;/script/gt;rwef6kpsx8n was submitted in the username parameter. This input was echoed unmodified in the application's response.This proof-of-concept attack demonstrates that it is possible to inject arbitrary JavaScript into the application's response.The original request used the POST method, however it was possible to convert the request to use the GET method, to enable easier demonstration and delivery of the attack. | |
| 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. | |
| Recommendation | |
| 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 /lt; /gt; " ' and =, should be replaced with the corresponding HTML entities (/amp;lt; /amp;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. | |
| References | |
|  | |

## Medium Severity Findings

## Low Severity Findings

|  |  |
| --- | --- |
| Unencrypted communications | |
| Severity: Low | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://detectportal.firefox.com | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://detectportal.firefox.com/ |  |
| Vulnerability Details | |
|  | |
| The application allows users to connect to it over unencrypted connections. An attacker suitably positioned to view a legitimate user's network traffic could record and monitor their interactions with the application and obtain any information the user supplies. Furthermore, an attacker able to modify traffic could use the application as a platform for attacks against its users and third-party websites. Unencrypted connections have been exploited by ISPs and governments to track users, and to inject adverts and malicious JavaScript. Due to these concerns, web browser vendors are planning to visually flag unencrypted connections as hazardous. 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. Please note that using a mixture of encrypted and unencrypted communications is an ineffective defense against active attackers, because they can easily remove references to encrypted resources when these references are transmitted over an unencrypted connection. | |
| Recommendation | |
| Applications should use transport-level encryption (SSL/TLS) to protect all communications passing between the client and the server. The Strict-Transport-Security HTTP header should be used to ensure that clients refuse to access the server over an insecure connection. | |
| References | |
|  | |
|  | |
|  | |
|  | |
| Password field with autocomplete enabled | |
| Severity: Low | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/index.php | GET |
| Vulnerability Details | |
| The page contains a form with the following action URL:http://s96141-102131-t7y.croto.hack.me/index.phpThe form contains the following password field with autocomplete enabled:password | |
| Most browsers have a facility to remember user credentials that are entered into HTML forms. This function can be configured by the user and also by applications that employ user credentials. If the function is enabled, then credentials entered by the user are stored on their local computer and retrieved by the browser on future visits to the same application. The stored credentials can be captured by an attacker who gains control over the user's computer. Further, an attacker who finds a separate application vulnerability such as cross-site scripting may be able to exploit this to retrieve a user's browser-stored credentials. | |
| Recommendation | |
| To prevent browsers from storing credentials entered into HTML forms, include the attribute autocomplete="off" within the FORM tag (to protect all form fields) or within the relevant INPUT tags (to protect specific individual fields). Please note that modern web browsers may ignore this directive. In spite of this there is a chance that not disabling autocomplete may cause problems obtaining PCI compliance. | |
| References | |
|  | |

## Informational Findings

|  |  |
| --- | --- |
| Cross-origin resource sharing | |
| Severity: Information | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://www.msn.com | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://www.msn.com/ | GET |
| Vulnerability Details | |
| The application implements an HTML5 cross-origin resource sharing (CORS) policy for this request.If the application relies on network firewalls or other IP-based access controls, this policy is likely to present a security risk.Since the Vary: Origin header was not present in the response, reverse proxies and intermediate servers may cache it. This may enable an attacker to carry out cache poisoning attacks. | |
| An HTML5 cross-origin resource sharing (CORS) policy controls whether and how content running on other domains can perform two-way interaction with the domain that publishes the policy. The policy is fine-grained and can apply access controls per-request based on the URL and other features of the request.If another domain is allowed by the policy, then that domain can potentially attack users of the application. If a user is logged in to the application, and visits a domain allowed by the policy, then any malicious content running on that domain can potentially retrieve content from the application, and sometimes carry out actions within the security context of the logged in user. Even if an allowed domain is not overtly malicious in itself, security vulnerabilities within that domain could potentially be leveraged by an attacker to exploit the trust relationship and attack the application that allows access. CORS policies on pages containing sensitive information should be reviewed to determine whether it is appropriate for the application to trust both the intentions and security posture of any domains granted access. | |
| Recommendation | |
| Any inappropriate domains should be removed from the CORS policy. | |
| References | |
|  | |
|  | |
| HTML does not specify charset | |
| Severity: Information | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/index.php | GET |
| Vulnerability Details | |
|  | |
| If a response states that it contains HTML content but does not specify a character set, then the browser may analyze the HTML and attempt to determine which character set it appears to be using. Even if the majority of the HTML actually employs a standard character set such as UTF-8, the presence of non-standard characters anywhere in the response may cause the browser to interpret the content using a different character set. This can have unexpected results, and can lead to cross-site scripting vulnerabilities in which non-standard encodings like UTF-7 can be used to bypass the application's defensive filters. In most cases, the absence of a charset directive does not constitute a security flaw, particularly if the response contains static content. You should review the contents of affected responses, and the context in which they appear, to determine whether any vulnerability exists. | |
| Recommendation | |
| For every response containing HTML content, the application should include within the Content-type header a directive specifying a standard recognized character set, for example charset=ISO-8859-1. | |
| References | |
|  | |
| Frameable response (potential Clickjacking) | |
| Severity: Information | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/favicon.ico | GET |
| Vulnerability Details | |
| If a page fails to set an appropriate X-Frame-Options or Content-Security-Policy HTTP header, it might be possible for a page controlled by an attacker to load it within an iframe. This may enable a clickjacking attack, in which the attacker's page overlays the target application's interface with a different interface provided by the attacker. By inducing victim users to perform actions such as mouse clicks and keystrokes, the attacker can cause them to unwittingly carry out actions within the application that is being targeted. This technique allows the attacker to circumvent defenses against cross-site request forgery, and may result in unauthorized actions. Note that some applications attempt to prevent these attacks from within the HTML page itself, using "framebusting" code. However, this type of defense is normally ineffective and can usually be circumvented by a skilled attacker. You should determine whether any functions accessible within frameable pages can be used by application users to perform any sensitive actions within the application. | |
| Recommendation | |
| To effectively prevent framing attacks, the application should return a response header with the name X-Frame-Options and the value DENY to prevent framing altogether, or the value SAMEORIGIN to allow framing only by pages on the same origin as the response itself. Note that the SAMEORIGIN header can be partially bypassed if the application itself can be made to frame untrusted websites. | |
| References | |
|  | |
| Cross-site request forgery | |
| Severity: Information | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/ | POST |
| Vulnerability Details | |
| The request appears to be vulnerable to cross-site request forgery (CSRF) attacks against unauthenticated functionality. This is unlikely to constitute a security vulnerability in its own right, however it may facilitate exploitation of other vulnerabilities affecting application users. | |
| 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. | |
| Recommendation | |
| 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. | |
| References | |
|  | |
| Input returned in response (reflected) | |
| Severity: Information | |
| CVSS Vector: 0.00 | |
| Remediation State*:* Open | |
| Affected Asset | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me | 80 |
| Affected URLs | |
| URL | PARAMETER(S) |
| http://s96141-102131-t7y.croto.hack.me/index.php | POST |
| Vulnerability Details | |
| The value of the username request parameter is copied into the application's response. | |
| Reflection of input arises when data is copied from a request and echoed into the application's immediate response.Input being returned in application responses is not a vulnerability in its own right. However, it is a prerequisite for many client-side vulnerabilities, including cross-site scripting, open redirection, content spoofing, and response header injection. Additionally, some server-side vulnerabilities such as SQL injection are often easier to identify and exploit when input is returned in responses. In applications where input retrieval is rare and the environment is resistant to automated testing (for example, due to a web application firewall), it might be worth subjecting instances of it to focused manual testing. | |
| Recommendation | |
|  | |
| References | |
|  | |