CIRR_PING_AUTH_MODULE Scan Report

Project Name	CIRR_PING_AUTH_MODULE
Scan Start	Friday, October 18, 2019 5:22:59 PM
Preset	Checkmarx Default
Scan Time	00h:01m:28s
Lines Of Code Scanned	11893
Files Scanned	28
Report Creation Time	Friday, October 18, 2019 5:24:30 PM
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446
Team	Cirrus
Checkmarx Version	8.9.0.210 HF1
Scan Type	Incremental
Source Origin	LocalPath
Density	7/10000 (Vulnerabilities/LOC)
Visibility	Public

Filter Settings

Severity

Included: High, Medium, Low, Information

Excluded: None

Result State

Included: Confirmed, Not Exploitable, To Verify, Urgent, Proposed Not Exploitable

Excluded: None

Assigned to

Included: All

Categories

Included:

Uncategorized	All
Custom	All
PCI DSS v3.2	All
OWASP Top 10 2013	All
FISMA 2014	All
NIST SP 800-53	All
OWASP Top 10 2017	All
OWASP Mobile Top 10 2016	All

Excluded:

Uncategorized	None
Custom	None
PCI DSS v3.2	None
OWASP Top 10 2013	None
FISMA 2014	None

NIST SP 800-53	None
OWASP Top 10 2017	None
OWASP Mobile Top 10 2016	None

Results Limit

Results limit per query was set to 50

Selected Queries

To see the selected queries you must check the 'Executive Summary' option in the 'General' section of the report template

Scan Results Details

Security Misconfiguration

Query Path:
JavaScript\Cx\JavaScript Server Side Vulnerabilities\Security Misconfiguration Version:2

Description

Security Misconfiguration\Path 1:

Severity	High
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=4
Status	Recurrent

The application takes sensitive, personal data cookieSecret, found at line 17 of src/endpoints.js, and stores it in an unprotected manner, without encryption, to session at line 17 of src/endpoints.js.

	Source	Destination
File	src/endpoints.js	src/endpoints.js
Line	24	24
Object	cookieSecret	session

Code Snippet
File Name src/endpoints.js
Method export const applyMiddleware = (router, authEnabled, authMiddleware) => {

```
....
24.      router.use(session({ secret: cookieSecret, resave: false,
saveUninitialized: false }));
```

Missing HSTS Header

Query Path:
JavaScript\Cx\JavaScript Medium Threat\Missing HSTS Header Version:1

Description

Missing HSTS Header\Path 1:

Severity	Medium
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=6
Status	Recurrent

The web-application does not define an HSTS header, leaving it vulnerable to attack.

	Source	Destination
File	dist/endpoints.js	dist/endpoints.js
Line	86	86
Object	json	json

Code Snippet
File Name dist/endpoints.js

Method	router.get(useridCallbackPath, function (req, res) {

```
....
86.       res.json({
```

Missing HSTS Header\Path 2:

Severity	Medium
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=8
Status	Recurrent

The web-application does not define an HSTS header, leaving it vulnerable to attack.

	Source	Destination
File	src/endpoints.js	src/endpoints.js
Line	70	70
Object	json	json

Code Snippet	
File Name	src/endpoints.js
Method	router.get(useridCallbackPath, (req, res) => {

```
....
70.       res.json({
```

Client DOM Open Redirect

Query Path:
JavaScript\Cx\JavaScript Low Visibility\Client DOM Open Redirect Version:2

Categories

OWASP Top 10 2013: A10-Unvalidated Redirects and Forwards
FISMA 2014: System And Information Integrity
NIST SP 800-53: SI-10 Information Input Validation (P1)

Description

Client DOM Open Redirect\Path 1:

Severity	Low
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=1
Status	Recurrent

A possible open redirect has been found at line 1 in dist/authClientSide.html file. This might lead to an untrusted site which mainly used for phishing.

	Source	Destination
File	dist/authClientSide.html	dist/authClientSide.html
Line	2	2
Object	replace	location

Code Snippet	
File Name	dist/authClientSide.html
Method	<script>

```
....
2.  window.location.href = window.location.href.replace(/#/, '?');
```

Client DOM Open Redirect\Path 2:

Severity	Low
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=2
Status	Recurrent

A possible open redirect has been found at line 1 in src/authClientSide.html file. This might lead to an untrusted site which mainly used for phishing.

	Source	Destination
File	src/authClientSide.html	src/authClientSide.html
Line	2	2
Object	replace	location

Code Snippet	
File Name	src/authClientSide.html
Method	<script>

```
....
2.  window.location.href = window.location.href.replace(/#/, '?');
```

Missing CSP Header

Query Path:
JavaScript\Cx\JavaScript Server Side Vulnerabilities\Missing CSP Header Version:1

Description

Missing CSP Header\Path 1:

Severity	Low
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=5
Status	Recurrent

A Content Security Policy is not explicitly defined within the web-application.

	Source	Destination
File	dist/endpoints.js	dist/endpoints.js
Line	78	78
Object	sendFile	sendFile

Code Snippet	
File Name	dist/endpoints.js

Method	router.use(openidCallbackPath, function (req, res, next) {

```
....
78.         return res.sendFile(clientAuth);
```

Missing CSP Header\Path 2:

Severity	Low
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=7
Status	Recurrent

A Content Security Policy is not explicitly defined within the web-application.

	Source	Destination
File	src/endpoints.js	src/endpoints.js
Line	61	61
Object	sendFile	sendFile

Code Snippet	
File Name	src/endpoints.js
Method	router.use(openidCallbackPath, (req, res, next) => {

```
....
61.         return res.sendFile(clientAuth);
```

Client Insufficient ClickJacking Protection

Query Path:
JavaScript\Cx\JavaScript Low Visibility\Client Insufficient ClickJacking Protection Version:5

Categories

FISMA 2014: Configuration Management
NIST SP 800-53: SC-8 Transmission Confidentiality and Integrity (P1)

Description

Client Insufficient ClickJacking Protection\Path 1:

Severity	Low
Result State	To Verify
Online Results	https://cxlilly.checkmarx.net/CxWebClient/ViewerMain.aspx?scanid=1074271&projectid=446&pathid=3
Status	Recurrent

The application does not protect the web page dependency_security_results.html from clickjacking attacks in legacy browsers, by using framebusting scripts.

	Source	Destination
File	dependency_security_results.html	dependency_security_results.html
Line	1	1
Object	CxJSNS_107981890	CxJSNS_107981890

Code Snippet
File Name dependency_security_results.html
Method <!DOCTYPE html>

```
....
1.  <!DOCTYPE html>
```

Security Misconfiguration

Risk

What might happen

It is possible that an attacker may, at some point, succeed in accessing the server. When doing so, the attacker may be able to read the server's memory, database, and such. Any sensitive information would then be revealed to the attacker, including users' personal details. This includes their account information, financial data, SSN, and more. Using this information, the attacker may be able to perform identity theft, or simply misuse the victims' private data.

Cause

How does it happen

The application stores sensitive, personal information, such as PII, in user sessions or the application database. This data is stored in plaintext, without encryption, allowing anyone with read access to the server to steal these secrets.

General Recommendations

How to avoid it

- Do not store personal data or other secrets in plain text, without encryption. This applies both to long-term storage such as a database, and even mid-term memory such as server-side sessions.
- Sensitive data should always be stored in an encrypted manner, using modern encryption algorithms (e.g. AES) and a sufficiently long encryption key (e.g. 256 bits). Encryption keys must also be protected.
- Alternatively, some types of data should not be stored in a reversible format at all, and can be hashed using a cryptographically strong hash algorithm, such as SHA-256.

Source Code Examples

JavaScript

Hardcoded Session Token using Express

```
var express = require('express');
var app = express();

app.use(express.cookieParser());
app.use(express.session({secret: '0123456789ABCDEF'}));
```

```

```

Protect Session Token in Config File using Express

```
var express = require('express');
var app = express();

app.use(express.cookieParser());

var token = decryptTokenFromConfigFile();

app.use(express.session({ secret: token,
                          cookie: {httpOnly: true,
                                   secure: true,
                                   maxAge: 1200000},
                          saveUninitialized: false
                        }));
```

PII Stored in Session Unencrypted with SessionJS

```
var http = require('http');
var session = require('./lib/core').session;
var querystring = require('querystring');

http.createServer(function (request, response) {
    session(request, response, function(request, response){
        var params = querystring.parse(request.url);
        var accountId = params.accountId;

        // Store the user's account details unencrypted
        request.session.data.accountId = accountId;
    });
}).listen(8080);
```

Encrypting PII before Storing with SessionJS

```
var http = require('http');
var session = require('./lib/core').session;
var querystring = require('querystring');

var crypto = require("crypto-js");

http.createServer(function (request, response) {
    session(request, response, function(request, response){
        var params = querystring.parse(request.url);
        var accountId = encrypt(params.accountId);

        // Always encrypt sensitive and personal data
        //   before storing it in long term memory or storage
        var encrypted = crypto.AES.encrypt(accountId, getEncKeyFromConfig());
        request.session.data.accountId = encrypted;
    });
}).listen(8080);
```

Missing HSTS Header

Risk

What might happen

Failure to set an HSTS header and provide it with a reasonable "max-age" value of at least one year may leave users vulnerable to Man-in-the-Middle attacks.

Cause

How does it happen

Many users browse to websites by simply typing the domain name into the address bar, without the protocol prefix. The browser will automatically assume that the user's intended protocol is HTTP, instead of the encrypted HTTPS protocol.

When this initial request is made, an attacker can perform a Man-in-the-Middle attack and manipulate it to redirect users to a malicious web-site of the attacker's choosing. To protect the user from such an occurence, the HTTP Strict Transport Security (HSTS) header instructs the user's browser to disallow use of an unsecure HTTP connection to the the domain associated with the HSTS header.

Once a browser that supports the HSTS feature has visited a web-site and the header was set, it will no longer allow communicating with the domain over an HTTP connection.

Once an HSTS header was issued for a specific website, the browser is also instructed to prevent users from manually overriding and accepting an untrusted SSL certificate for as long as the "max-age" value still applies. The recommended "max-age" value is for at least one year in seconds, or 31536000.

General Recommendations

How to avoid it

- Before setting the HSTS header - consider the implications it may have:
  - Forcing HTTPS will prevent any future use of HTTP, which could hinder some testing
  - Disabling HSTS is not trivial, as once it is disabled on the site, it must also be disabled on the browser
- Set the HSTS header either explicitly within application code, or using web-server configurations.
- Ensure the "max-age" value for HSTS headers is set to 31536000 to ensure HSTS is strictly enforced for at least one year.
- Once HSTS has been enforced, submit the web-application's address to an HSTS preload list - this will ensure that, even if a client is accessing the web-application for the first time (implying HSTS has not yet been set by the web-application), a browser that respects the HSTS preload list would still treat the web-application as if it had already issued an HSTS header. Note that this requires the server to have a trusted SSL certificate, and issue an HSTS header with a maxAge of 1 year (31536000)
- Note that this query is designed to return one result per application. This means that if more than one vulnerable response without an HSTS header is identified, only the first identified instance of this issue will be highlighted as a result. Since HSTS is required to be enforced across the entire application to be considered a secure deployment of HSTS functionality, fixing this issue only where the query highlights this result is likely to produce subsequent results in other sections of the application; therefore, when adding this header via code, ensure it is uniformly deployed across the entire application. If this header is added via configuration, ensure that this configuration applies to the entire application.

Source Code Examples

JavaScript
Using Helmet with Express

```
var express = require('express')
var helmet = require('helmet') // Helmet includes HSTS as a built-in header

var app = express()
app.use(helmet())
```

Using Explicit HSTS Package (Built into Helmet, Both Are Not Required)

```
var hsts = require('hsts')

app.use(hsts({
  maxAge: 31536000
}))
```

Explicitly Setting HSTS Header in Code

```
res.setHeader("Strict-Transport-Security", "max-age=31536000");
```

Client DOM Open Redirect

Risk

What might happen

An attacker could use social engineering to get a victim to click a link to the application, so that the user will be immediately redirected to another, arbitrary site. Users may think that they are still in the original application site. The second site may be offensive, contain malware, or, most commonly, be used for phishing.

Cause

How does it happen

The application redirects the user's browser to a URL provided in a user request, without warning users that they are being redirected outside the site. An attacker could use social engineering to get a victim to click a link to the application with a parameter defining another site to which the application will redirect the user's browser, and the user may not be aware of the redirection.

General Recommendations

How to avoid it

1. Ideally, do not allow arbitrary URLs for redirection. Instead, create a server-side mapping from user-provided parameter values to legitimate URLs.
2. If it is necessary to allow arbitrary URLs:
    - For URLs inside the application site, first filter and encode the user-provided parameter, and then use it as a relative URL by prefixing it with the application site domain.
    - For URLs outside the application (if necessary), use an intermediate disclaimer page to provide users with a clear warning that they are leaving your site.

Source Code Examples

CSharp

Avoid redirecting to arbitrary URLs, instead map the parameter to a list of static URLs.

```
Response.Redirect(getUrlById(targetUrlId));
```

Java

Avoid redirecting to arbitrary URLs, instead map the parameter to a list of static URLs.

```
Response.Redirect(getUrlById(targetUrlId));
```

Apex

Client Insufficient ClickJacking Protection

Risk

What might happen

Clickjacking attacks allow an attacker to "hijack" a user's mouse clicks on a webpage, by invisibly framing the application, and superimposing it in front of a bogus site. When the user is convinced to click on the bogus website, e.g. on a link or a button, the user's mouse is actually clicking on the target webpage, despite being invisible.
This could allow the attacker to craft an overlay that, when clicked, would lead the user to perform undesirable actions in the vulnerable application, e.g. enabling the user's webcam, deleting all the user's records, changing the user's settings, or causing clickfraud.

Cause

How does it happen

The root cause of vulnerability to a clickjacking attack, is that the application's web pages can be loaded into a frame of another website. The application does not implement a proper frame-busting script, that would prevent the page from being loaded into another frame. Note that there are many types of simplistic redirection scripts that still leave the application vulnerable to clickjacking techniques, and should not be used.
When dealing with modern browsers, applications mitigate this vulnerability by issuing appropriate Content-Security-Policy or X-Frame-Options headers to indicate to the browser to disallow framing. However, many legacy browsers do not support this feature, and require a more manual approach by implementing a mitigation in Javascript. To ensure legacy support, a framebusting script is required.

General Recommendations

How to avoid it

Generic Guidance:

- Define and implement a a Content Security Policy (CSP) on the server side, including a frame-ancestors directive. Enforce the CSP on all relevant webpages.
- If certain webpages are required to be loaded into a frame, define a specific, whitelisted target URL.
- Alternatively, return a "X-Frame-Options" header on all HTTP responses. If it is necessary to allow a particular webpage to be loaded into a frame, define a specific, whitelisted target URL.
- For legacy support, implement framebusting code using Javascript and CSS to ensure that, if a page is framed, it is never displayed, and attempt to navigate into the frame to prevent attack. Even if navigation fails, the page is not displayed and is therefore not interactive, mitigating potential clickjacking attacks.

Specific Recommendations:

- Implement a proper framebuster script on the client, that is not vulnerable to frame-buster-busting attacks.
    - Code should first disable the UI, such that even if frame-busting is successfully evaded, the UI cannot be clicked. This can be done by setting the CSS value of the "display" attribute to "none" on either the "body" or "html" tags. This is done because, if a frame attempts to redirect and become the parent, the malicious parent can still prevent redirection via various techniques.
    - Code should then determine whether no framing occurs by comparing self === top; if the result is true, can the UI be enabled. If it is false, attempt to navigate away from the framing page by setting the top.location attribute to self.location.

Source Code Examples

JavaScript
Clickjackable Webpage

```
<html>
    <body>
     <button onclick="clicked();">
            Click here if you love ducks
        </button>
    </body>
</html>
```

Bustable Framebuster

```
<html>
    <head>
     <script>
            if ( window.self.location != window.top.location ) {
                window.top.location = window.self.location;
            }
        </script>
    </head>

    <body>
        <button onclick="clicked();">
            Click here if you love ducks
        </button>
    </body>
</html>
```

Proper Framebusterbusterbusting

```
<html>
    <head>
     <style> html {display : none; } </style>
        <script>
            if ( self === top ) {
                document.documentElement.style.display = 'block';
            }
            else {
                top.location = self.location;
            }
        </script>
    </head>

    <body>
        <button onclick="clicked();">
            Click here if you love ducks
        </button>
    </body>
</html>
```

Missing CSP Header

Risk

What might happen

The Content-Security-Policy header enforces that the source of content, such as the origin of a script, embedded (child) frame, embedding (parent) frame or image, are trusted and allowed by the current web-page; if, within the web-page, a content's source does not adhere to a strict Content Security Policy, it is promptly rejected by the browser. Failure to define a policy may leave the application's users exposed to Cross-Site Scripting (XSS) attacks, Clickjacking attacks, content forgery and more.

Cause

How does it happen

The Content-Security-Policy header is used by modern browsers as an indicator for trusted sources of content, including media, images, scripts, frames and more. If these policies are not explicitly defined, default browser behavior would allow untrusted content.

General Recommendations

How to avoid it

Explicitly set the Content-Security-Policy headers for all applicable policy types (frame, script, form, script, media, img etc.) according to business requirements and deployment layout of external file hosting services. Specifically, do not use a wildcard, '*', to specify these policies, as this would allow content from any external resource.

The Content-Security-Policy can be explicitly defined within web-application code, as a header managed by web-server configurations, or within <meta> tags in the HTML `<head>` section.

Source Code Examples

PHP

Restricting Content-Security-Policy to Only Obtain Embedded Content from Current Web-Application

```
<?php
      header("Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src
'self'; img-src 'self'; style-src 'self';");
?>
```

Scanned Languages

Language	Hash Number	Change Date
JavaScript	1003522720031683	6/2/2019
VbScript	9340222351170833	6/2/2019
Typescript	1488217042171263	6/2/2019
Common	0114668597102001	6/2/2019