Findings Report

OWASP Top 10 2021 - Demo

COMMISSIONED BY DEMO - CLIENT

EXECUTION DATES : 04-10-2021 - 17-11-2021

REPORT STATUS: Final

VERSION : 1.0

DATE : 24-01-2022

REVISION INFORMATION:

EXECUTED BY: Mike Terhaar | Senior Ethical Hacker for CYVER BV

THIS REPORT IS CONFIDENTIAL

1. MANAGEMENT SUMMARY

1.1. GENERAL INFORMATION

A penetration test is a controlled attack on a computer or network system with intent to find security weaknesses and potentially gain access to the system and its data. The process involves identifying target systems and setting a goal for the attack, obtaining contextual and technical information, followed by phases of vulnerability identification and validation. Potential solutions are then shared to help the organization mitigate any discovered vulnerabilities.

During the penetration test, the pentester adopts the mindset and approach of an attacker. This allows us to assess the infrastructure from the point of view of a real-world attacker's. This greatly differs from the perspective of operational teams implementing and operating IT solutions. The shift to new perspectives also creates insight into information available to an attacker and vulnerabilities, which might be discovered and exploited.

Ultimately, the goal is that the organization conducting a pentest can reduce risk in a focused and cost-effective manner. Penetration tests also help in determining where general weaknesses exist in the digital environment, which leads to a more proactive approach in security management processes. For example, when control mechanisms are built into the policies and procedures driving daily operations.

The main objective of the penetrating test is to provide input from a technical and real-world perspective. This input is designed to be valuable in helping the organization to determine and reduce business risk. We obtain that information by identifying system and infrastructure vulnerabilities, which could be used by an attacker to gain unauthorized access to systems or information, while providing sufficient information for the organization to understand associated business risks and potential impact of those vulnerabilities - so that the organization can assess actions and prioritize how to mitigate risks.

This report covers the results of the OWASP Top 10 2021 - Demo security investigation conducted by CYVER at the request of DEMO - CLIENT.

This security investigation was conducted from 04-10-2021 - 17-11-2021.

The research methodology used for a Web Security Assessment follows the guidelines in the OWASP Testing Guide.

1.2. SCOPE AND APPROACH

This report covers the results of the following scope information:

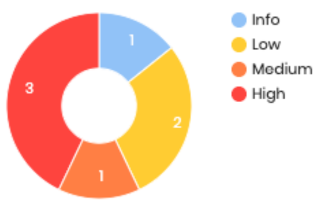
Asset	Type
Cyver Demo App [https://cyver-demo-app.io]	Web Application

This pentest was conducted from 07-07-2021 - 09-07-2021.

The timeframe of the test is 1 9:10 - 17:12 (WSTG items 1-6) Day 2 8:00 - 17:43 (WSTG 7-11) in accordance with research standards including PTES, OWASP Testing Guide.

1.2. RESULTS

The pentest identified a number of vulnerabilities. Found vulnerabilities are detailed below, organized by risk classification.



The research allowed us to form a clear impression of the investigated systems and/or applications.

1.3. RECOMMENDATIONS

Based on the findings, CYVER has the following recommendations:

1. Research has shown that Cross-Site scripting is possible. This is considered a serious vulnerability, which should be fixed immediately. Our advice is to take immediate mitigating actions by adjusting IPS configuration so that XSS can be detected and stopped. In addition, improving the CSP policy can help to reduce visibility. We also recommend adjusting the input/output filters so that XSS is no longer possible.
2. Implement protective measures such as installing or activating a Web Application Firewall. This is a Quick fix and Workaround, not a permanent fix. Step 2 is to mitigate the issue.
3. Solve the basic configuration issues
4. . . . .

1.4. CONCLUSION

This input depends on the assignment. There is a sample below.
Cyver had to prove that a new customer's application was secure. One precondition for demonstrating a secure application is that no high criticality findings from the OWASP top 10 2021 may be discovered. The test resulted in two high findings rated against the OWASP Top 10 A03:2021 Injection. This means that, according to the boundaries set for this project, the application did NOT PASS the requirements.

2. TECHNICAL SUMMARY

This chapter provides an overview of key findings included in this report. For a detailed description, see the actual findings later in this report.

2.1. Findings Summary

Vulnerability	Severity	CVSS 3.1
F-2021-0937 - Cross-site scripting (stored)	High	8.2
F-2021-0938 - Cross-site scripting (reflected)	High	7.6
F-2021-0939 - Form does not contain an anti-CSRF token	High	7.1
F-2021-0933 - CSP: Inline scripts can be inserted	Medium	5.3
F-2021-0934 - Client-side HTTP parameter pollution (reflected)	Low	3.7
F-2021-0941 - Strict transport security not enforced	Low	3.3

2.2. OWASP Reference

🟥 OWASP Top 10 2021 — 0 Critical · 3 High · 1 Medium · 2 Low · 1 Info

Control	Findings
▨ **A01:2021 | Broken Access Control**	
A01:2021 - Broken Access Control	0 Critical · 0 High · 0 Medium · 0 Low · 0 Info
▨ **A02:2021 | Cryptographic Failures**	
A02:2021 - Cryptographic Failures	0 Critical · 0 High · 0 Medium · 0 Low · 0 Info
🟥 **A03:2021 | Injection**	
A03:2021 - Injection	0 Critical · 2 High · 0 Medium · 1 Low · 0 Info
▨ **A04:2021 | Insecure Design**	
A04:2021 - Insecure Design	0 Critical · 0 High · 0 Medium · 0 Low · 0 Info
🟥 **A05:2021 | Security Misconfiguration**	
A05:2021 - Security Misconfiguration	0 Critical · 1 High · 1 Medium · 1 Low · 1 Info
▨ **A06:2021 | Vulnerable and Outdated Components**	
A06:2021 - Vulnerable and Outdated Components	0 Critical · 0 High · 0 Medium · 0 Low · 0 Info
▨ **A07:2021 | Identification and Authentication Failures**	
A07:2021 - Identification and Authentication Failures	0 Critical · 0 High · 0 Medium · 0 Low · 0 Info
▨ **A08:2021 | Software and Data Integrity Failures**	
A08:2021 - Software and Data Integrity Failures	0 Critical · 0 High · 0 Medium · 0 Low · 0 Info
▨ **A09:2021 | Security Logging and Monitoring Failures**	

Control	Findings
A09:2021 - Security Logging and Monitoring Failures	0 Critical 0 High 0 Medium 0 Low 0 Info
▨ **A10:2021 | Server Side Request Forgery (SSRF)**	
A10:2021 - Server Side Request Forgery (SSRF)	0 Critical 0 High 0 Medium 0 Low 0 Info

2.3. Main Findings

- Cross-site scripting (stored)
- Cross-site scripting (reflected)
- Form does not contain an anti-CSRF token

2.4. Observations

Observation
F-2021-0935 - Cross-domain Referer leakage

3. Assignment Details

This pentest is based on the agreed-upon proposal with reference xxxx, which includes the following description of the assignment:

1. . . .
2. . . .
3. . . .

3.2. Scope

The scope of the investigation is shown in the table below. Systems and applications not listed have not been pentested..

Asset	Type
Cyver Demo App [https://cyver-demo-app.io]	Web Application

3.3. Purpose of this Assignment

Independently determining the security level of OWASP Top 10 2021 - Demo , detecting vulnerabilities, and suggesting possible security improvements. The second goal is to create proof that the application is secure. In this case, application security is measured against the OWASP Top 10 2021. In case a high criticality vulnerability is discovered, the application fails the security assessment.

3.4. Research Method

This pentest was conducted using grey-box methodology. This means that the pentester has some credentials and some information about the test objects when starting the pentest.

Cyver used OWASP WSTG 4.2 for the assessment.

3.5. Reporting

The management summaries and technical summaries respectively serve as an overview of research results for general and technical management. In subsequent chapters, the detailed results are described and supported by reproducible findings. These chapters are intended to guide technical personnel in reproducing and mitigating vulnerability findings.

3.6. Limitations

A Pentest provides valuable insight into the IT security of the target system or application. However, this investigation is only a snapshot and does not guarantee the security of the IT environment and data. New attack techniques are constantly being developed and discovered. In addition, a small adjustment to the IT environment can easily introduce new vulnerabilities. The role of processes, procedures, and the human factor in information security are at least as important as technology used.

This report only provides an overview of vulnerabilities found and is therefore not intended as a guarantee of security.

In addition, it is important to note that a Pentest is performed by people and that during the research period, choices are made in approach and use of tooling. Pentest results also heavily rely on the capabilities of the executive consultant. Therefore, it is possible that tests with a repetitive nature may show different results.

4. APPLICATION RESEARCH

Cyver used the OWASP Web Security Testing Guide version 4.2 for this pentest. The sections in this chapter conform to OWASP WSTG classifications and designations. Sub-sections are not always recognizable by name, so more widely recognized, general terminology has sometimes been chosen. In addition, some vulnerability findings may relate to multiple sections of the WSTG. For example, the outcome of a port scan with NMAP on a vulnerability scan with Nessus. E.g., Port Scan (NMAP) can relate to "Information Gathering" as well as to "Configuration and Deployment".

4.1. Information Gathering

4.1.1 Architecture

Below is a graphic representation of the environment where the domain in question is hosted. The image gives a good overview, as seen from the Internet, of how the web application and the various connected services, such as the DNS, relate.

[image here]

4.1.2 Directory Indexing - SiteMap

The application has been assessed for easily findable data using the Gobuster application:

```
Gobuster v3.1.0
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     https://www.cyver.io/website/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                /Users/mike/Desktop/CYVER-PROJECT/directory-list-1.0.txt
[+] Negative Status codes:   404
[+] Cookies:                 s[...]
[+] User Agent:              gobuster/3.1.0
[+] Timeout:                 10s
===============================================================
2021/12/24 14:43:36 Starting gobuster in directory enumeration mode
===============================================================
/tv                   (Status: 302) [Size: 0] [--> https://www.cyver.io/website/sfmlkmg.scherm0404]
/toolbar              (Status: 302) [Size: 0] [--> https://www.cyver.io/website/sfmlkmg.scherm0404]
/cgi-bin              (Status: 302) [Size: 0] [--> https://www.cyver.io/website/sfmlkmg.scherm0404]
/-                    (Status: 302) [Size: 0] [--> https://www.cyver.io/website/sfmlkmg.scherm0404]
/talk                 (Status: 302) [Size: 0] [--> https://www.cyver.io/website/sfmlkmg.scherm0404]
/a-z                  (Status: 302) [Size: 0] [--> https://www.cyver.io/website/sfmlkmg.scherm0404]
```

In many cases, when visiting a page, the following was answered by the server:

```
[...]
<html ><head ><title >Request Rejected </title ></head ><body >The requested URL was rejected. Please consult wit
h your administrator.<br ><br >Your support ID is: 58072
```

No directory indexing capabilities were found.

4.2. Web Server Configuration and Deployment

This chapter describes the research into the infrastructure on which the web application is hosted. Resulting information was used to scan identified services in more detail, using the Nessus vulnerability scanner. Other specific tools such as OpenSSL have been used where relevant. This research was conducted on the production environment, from the perspective of a remote attacker on the Internet.

All identified vulnerabilities have been manually verified. Vulnerabilities are only included in this chapter if this check is positive. If services have not been identified or tested, the reason will be shared.

4.2.1. Enumeration and Port Scan IP

A port scan is a method of assessing which services are active on the host (enumeration) and then examining whether these services pose a threat to the service (web application). In a full penetration test, this phase includes an attempt to fool open ports and services and to exploit known weaknesses.

```
Password:
Starting Nmap 7.92 ( https://nmap.org ) at 2022-01-03 15:32 CET
Nmap scan report for cyver.io (91.184.0.79)
Host is up (0.016s latency).
rDNS record for 91.184.0.79: n5nj1sm.lb.shared.prod.hostnet.nl
Not shown: 907 filtered tcp ports (no-response), 90 closed tcp ports (conn-refused)
PORT    STATE SERVICE
21/tcp     open   ftp
80/tcp    open   http
443/tcp   open   https

Nmap done: 1 IP address (1 host up) scanned in 36.80 seconds
```

4.2.2. Vulnerability Scan NESSUS Pro

A Vulnerability scan was performed with Nessus Pro. The tooling was set up to check for known vulnerabilities at the IP and server level, as well as in the web environment. Findings are included in the general list of findings and are not discussed here.

4.2.3. HTTP Headers

4.2.3.1 HTTP Methods

The web server only supports POST, GET and OPTIONS. This is in line with best practice. In addition, debug methods like TRACE are not allowed..

```
Access-Control-Allow-Methods: GET,POST,HEAD,OPTIONS
```

4.2.3.2 HTTP Security Headers

This section discusses the HTTP security headers. For this purpose, reference was made to the following full response header:

```
Date: Mon , 31 Feb 1988 08:44:01 CMT
X-Frame -Options: SAMEORIGIN
Content -Security -Policy: frame -ancestors 'self';default -src 'self' 'unsafe -inline'*.youtube -nocookie.com; s
cript -src 'self' 'unsafe -inline' 'unsafe -eval' code.highcharts.com code.jquery.com; img -src 'self' data: blob
:i.ytimg.com img.youtube.com; style -src 'self' 'unsafe -inline' 'unsafe eval' data: blob: fonts.googleapis.com;
font -src 'self' data:; object -src 'none'; base -uri 'self'; HTTP Strict Transport Security
Content -Location: /xxxxxx /!ut/p/z1 /[...]
Cache -Control: no -cache , no -store , must -revalidate
Expires: Thu , 01 Jan 1970 00:00:00 CMT
Pragma: no -cache
Vary: Cookie ,User -Agent
Last -Modified: Mon , 21 Dec 2020 09:47:20 CMT
X-XSS -Protection: 1; mode=block
X-Content -Type -Options: nosniff
X-Permitted -Cross -Domain -Policies: none
Referrer -Policy: same -origin
Strict -Transport -Security: max -age =33136000; includeSubDomains; preload
Connection: close
Content -Type: text/html; charset=UTF -8
Content -Language: en
Content -Length: 50265
```

Content Type

Pentesters checked that server responses contain a 'Content-Type' header defining a safe character set (e.g. UTF-8, ISO 8859-1).

No problems were found, as this header was set for all responses.

```
Content -Type: text/html; charset=UTF -8
```

Content Security Policy

Applied CSP config:

```
*.youtube -nocookie.com; script -src 'self' 'unsafe -inline' 'unsafe -eval' code.highcharts.com code.jquery.com;
img -src 'self' data: blob: i.ytimg.com img.youtube.com; style -src 'self' 'unsafe -inline' 'unsafe -eval' data:
blob: fonts.googleapis.com; font -src 'self' data:; object -src 'none'; base -uri 'self';
```

The assessment shows that the Content-Security-Policy-header's default-src and script-src definitions use the expression `unsafe-inline`.

`Unsafe-inline` instructs the browser to execute inline JavaScript code.

In addition, the Content-Security-Policy-header assessment shows the script-src-definitions use `unsafe-eval`. The use of `unsafe-eval` removes the sandbox function from the CSP header.

Style-src definitions also use `unsafe-inline` and `unsafe-eval` values. In addition, Frame-src was missing. This sets the default-SRC value, which contains unsafe-inline.

The risk with this is that an attacker is able to abuse any XSS attack vector that was possible before applying CSP. Therefore, CSP has no added value in the current configuration. The `unsafe-eval` expression allows scripts to evaluate strings as JavaScript.

Our advice is to not to make use of `unsafe-inline` and `unsafe-eval` in CSP or to use a nonce of at least 8 characters.

Content Type Options

The X-Content-Type-Options: nosniff header prevents browsers from determining the file MIME type for opening it. MIME confusion attacks can be prevented by disallowing this. The application defines this header correctly:

```
X-Content -Type -Options: nosniff
```

Strict Transport Security

HTTP Strict Transport Security (HSTS) was used to ensure TLS encryption of the connection:

```
Strict -Transport -Security: max -age =33136000; includeSubDomains; preload
```

Users accessing the site over HTTP are automatically redirected to the HTTPS environment using a HTTP 30X status code.

Referrer Headers

The browser sends a Referrer header when following a link within the application to another website. This header indicates which page the user came from. This allows potentially sensitive information from the URL to be redirected to other websites. The following Referrer-Policy-header was set within the application:

```
Referrer -Policy: same -origin
```

This header ensures the Referrer header is only sent within its own application and is therefore configured correctly.

X-Frame-Options - Clickjacking

The X-Frame-Options was added to HTTP responses from the server:

```
X-Frame -Options: SAMEORIGIN
```

On modern systems, this header prevents a frame from being included in another site, therefore preventing clickjacking. Therefore, it is properly configure..

4.2.4. Administrative Interfaces

No administrative interfaces were found.

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT]

4.3. Identity Management Testing

[SAMPLE REPORT - THESE CHAPTERS WILL BE INCLUDED IN THE ACTUAL REPORT]

4.4. Authentication Testing

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT]

4.4.1. Login

The login page was located at URL https://www.cyver.nl. The user was redirected to a SAML request authentication page:

No findings related to the login process were identified.

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT]

4.4.2 Logout

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT]

4.5. Authorization Testing

This section discusses the assessment of required authorization control mechanisms.

4.5.1. Bypassing Authorization Schema

Two standard users were created for the test.

Separate requests were performed for each loading the page and retrieving sensitive information. Requests were performed with the users' cookies to attempt to retrieve this sensitive information.

- TestuserCyver4
- TestuserCyver5

An example of a request is one for retrieving a user's personal information. For this, the following request was sent:

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT]

As shown above, the information was retrieved based on the user's session. As a result, it was not possible to access any other user's data.

4.6. Session Management

The application's session management mechanisms were assessed.

4.6.1. Session Management

Cookies and other session tokens should not appear in page URLs. In addition they should not appear in error messages from the server. These tokens should always be sent in HTTP headers or bodies with appropriate security measures.

In no case did the application display session identifiers in query string parameters or error messages.

4.6.2. Cookies Attributes

The following cookies were set by the application:

```
Set -Cookie: TS0189a202 =0[...]d; Path=/; Secure; HTTPOnly
Set -Cookie: JSESSIONID =0[...]1; Path=/; Secure; HttpOnly; SameSite=None; Secure;HttpOnly
Set -Cookie: !Proxy!proxyJSESSIONID=V[...]0; Path=/wps/eforms/proxy; Secure;
SameSite=None; Secure; HttpOnly
Set -Cookie: WASReqURL=h[...]/; Path=/; Secure; HttpOnly; SameSite=None; Secure;
```

The cookie `LtpaToken2` contained the session identifier. When this cookie was omitted, the previously logged in user had to log in again.

The `HttpOnly` parameter was set for all cookies containing sensitive information.

The `Secure` parameter was set for all cookies used in the HTTPS environment.

The `SameSite` parameter was set for the session cookie, but it was set to `None`. This caused the session cookie to be sent on cross-site requests. This lacks a depth defense measure against CSRF attacks.

4.6.3. Session Fixation

If an application does not refresh the set session token after successfully authenticating a user, it may allow an attacker to force the use of a predetermined token. If this token is known to the attacker, it may give the attacker access to the user's session.

After user authentication, the application refreshes the session identifier:

```
Date: Mon , 31 Feb 1988 09:19:06 CMT
Location: https :// wwwacc.mijncyver.nl/myportal/naar -mijncyver
Set -Cookie: LtpaToken2=l[...]
Set -Cookie: TS0189a202 =0[...]
[...]
```

As a result, this component is not vulnerable to session fixation attacks.

4.6.4. Session Variables

Session data was not stored at the client. Instead, the client kept a session identifier corresponding to session data managed by the server.

A session identifier should be unpredictable, meaning that it should be sufficiently random.

During the pentest, it was not possible to use the Burp-Suite sequencer to assess the session identifier. Therefore, the entropy of the session cookie was not tested.

4.6.5. Cross Site Request Forgery

An application should take strong measures to prevent CSRF attacks. . A successful CSRF attack can lead to compromised user data and actions, by forcing a legitimate user to perform actions specified by the attacker. With CSRF, it is possible to trick an authenticated user so that an attacker can access sensitive data.

Each request that performed an action in the application contained an unpredictable value. If this value was changed or omitted, the application refused to perform the operation. This prevents CSRF attacks. The value was in the URL for performing an action. In the case of removing a specific mobile, a GET request was sent to

[SAMPLE REPORT - THESE CHAPTERS IS INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]=

No vulnerabilities related to CSRF were identified.

4.6.6. Session Timeout

Session termination is a crucial part of session management. The validity period of a session identifier should be kept short as allowed by usability and application logic requirements.

Secure session termination requires the following three components:

- Availability of a manual logout functionality, accessible from the graphical user interface.
- Automatic session expiration after a specified time.

- Revocation of the validity of a terminated session on the server, or deletion of all session secrets on the client.

A clearly visible logout functionality was present on every page in the application. This reduces the likelihood of a user leaving the application without logging out. The time interval in which an attacker can abuse a compromised session is therefore kept to a minimum.

The logout link in the browser was activated. The session identifier was then restored in the browser; it was not possible to continue using the application's functionality in the context of an authenticated user without first logging in again. The session was correctly validated on the server. For more information, see the logout process in section 4.4.2.

Next, session expiration was examined by leaving an unused session open in a browser window. After no more than fifteen minutes of inactivity, the session was unusable. This limits the time interval in which an attacker can abuse a compromised session.

4.6.8. Session Puzzling

Session Variable Overloading (also known as Session Puzzling) is an application-level vulnerability that could allow an attacker to perform multiple malicious actions.

This security vulnerability occurs when an application uses the same session variable for more than one purpose. An attacker could potentially access pages in an order unforeseen by the developers, so that the session variable is cached in one context and then used in …

4.6.9. Session Hijacking

An attacker who gains access to cookies from user sessions can copy them and present these cookies to the web application. This attack is known as session hijacking. This can be prevented by setting the flag `secure`.

The following cookies were set by the application:

```
Set -Cookie: TS0189a202 =0[...]d; Path=/; Secure; HTTPOnly
Set -Cookie: JSESSIONID =0[...]1; Path=/; Secure; HttpOnly; SameSite=None; Secure;HttpOnly
Set -Cookie: !Proxy!proxyJSESSIONID=V[...]0; Path=/wps/eforms/proxy; Secure;
SameSite=None; Secure; HttpOnly
Set -Cookie: WASReqURL=h[...]/; Path=/; Secure; HttpOnly; SameSite=None; Secure;
```

It is positive to see that the `secure` parameter is set on all present cookies
In addition, HSTS is correctly configured to ensure TLS encryption.

```
Strict -Transport -Security: max -age =33136000; includeSubDomains; preload
```

Users accessing the site over HTTP are automatically redirected to the HTTPS environment using an HTTP 30X status code.

4.7. Input Validation, Sanitization and Coding

The most common security problems relate to failure to effectively validate input before it is processed or displayed. Nonvalidated input might be user input or from other sources in the system environment. However, the lack of effective input validation can result in vulnerabilities to XSS and SQL injection or other attacks.

4.7.1. Input Validation

Effective input validation protects against vulnerabilities such as parameter pollution. In addition, it can also be a mitigating factor for various injection attacks. An application can also take a number of proactive measures to maintain effective input validation. The following techniques contribute to this:

- Positive validation: using a 'so-called' whitelist instead of a blacklist to determine which input is valid and which is not.
- Strict enforcement of data types: validating data using prior knowledge about the expected data type and format (IBAN, address, mobile number, and so on)
- Validate URLs and redirects: the application should not allow redirects to external, untrusted pages.

4.7.1.1. Sanitizing Data Types

The assessment of Cyver's application revealed that information was strictly controlled by data type.

[SAMPLE REPORT - THESE CHAPTERS IS INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF];

Based on this assessment, we must conclude that input validation security is inadequate. Input validity is not checked consistently. Mobile Numbers and auto-completed fields such as street and city are not checked for validity when they are sent from the user. It was also possible to change several values to strings with a length of fifty thousand characters.

The recommendation is to validate all input coming from the user, including fields that are not normally visible in the user interface.

4.7.2. Output Validation

Encrypting output is critical for the security of a web application. Including untrusted input can result in it being seen and executed as code.

4.7.3. Cross-site scripting

HTML meta-characters were included in the page to test for XSS. This was possible, for example, in the cyver message box. Here the '<' and '>' characters were placed in the message:

```
Host: wwwacc.mijncyver.nl
```

```
[...]
```

```
[...]& street=xxx+xxx <>&city=MIAMI <>&street =&city =& emailAdres=jason.co%40 cyver.nl&mobile1Display =%2 B31 +%
```

```
280%29+6123456788& mobile1 =0033112345678& mobile2Display =%2 B31+%280%29+651273283& mobile2 =0033151273283& homea
rtsId =94072& apotheekId =44450
```

In doing so, the following was displayed in the personal data summary:

```
<span class=" locality" data -bind="text: addressObject.city">MIAMI<> </span >
```

As shown, the HTML characters were correctly HTML-encoded. In short characters < and > are encoded in < and >, preventing javascript. Similar tests had been performed throughout the application. No XSS vulnerabilities were identified in the application.

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

4.7.4. Output

Encrypting output is critical for web application security. Including untrusted input can result in it being seen and executed as code.

4.7.6. SQL Injection

Customizing personal data has been tested for SQL injection.

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

4.8. Error Handling

Error Messages are usually considered benign because they contain diagnostic data and messages that can help the user understand the problem or for the developer to debug that error.

Attackers can send unexpected data or force the system into certain edge cases and scenarios in an attempt to get the system or application to disclose that data. This works, unless developers have eliminated all possible errors and set a generic or custom message instead.

When an unexpected error occurred during the test, general error pages containing no internal information were displayed.

```
[...]
<html ><head ><title >Request Rejected </title ></head ><body >The requested URL was rejected. Please consult wit
h your administrator.<br ><br >Your support ID is: 5807284741286 <br ><br ><a href='javascript:history.back();' >
[Go Back]</a></body ><
/html >
```

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

It is clear from the responses that no internal or technical information, which could provide a potential attacker with information about the environment, is revealed.

4.9. Cryptography

4.9.1. TLS

When information is transmitted between the client and the server, it must be encrypted and secured to prevent an attacker from reading or modifying it. This is usually done using HTTPS, which uses the Transport Layer Security (TLS) protocol, a replacement for the older Secure Socket Layer (SSL) protocol. TLS also provides the server with a way to demonstrate to the client that they have connected to the correct server by presenting a trusted digital certificate.

Encryption and encryption is not always well understood. Cyver uses the table below, which provides an overview of the potential TLS vulnerabilities that have been investigated. For each server, it indicates which vulnerabilities have been mitigated.

TLS is examined for at least the following information:

- Server certificate validity
- Revocation certificate present
- Name of the certificate CN/AN
- Certificate not revoked
- Disabled SSLv2
- Disabled SSLv3
- Disabled TLSv1.0
- Support for TLSv1.2 or newer
- Vulnerabilities such as Heartbleed, POODLE, Freak, downgrade attack etc.
- No signatures like MD5 / SHA-1
-

4.9.1.1 Certificate Chain and Server Certificate

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

4.9.1.2 Cyphers

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

4.9.2. Sensitive Information Sent via Unencrypted Channels

If the application sends sensitive information via unencrypted channels - e.g. HTTP - it is considered a security risk. Some examples include basic authentication where plain text authentication data is sent over HTTP, form-based authentication data sent over HTTP, or plain text transmission of other information considered sensitive due to regulations, laws, organizational policies, or application business logic.

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

During the test, there was no reason to believe that sensitive information was available over an insecure channel.

4.10. Business Logic

Testing for business logic errors in a multifunctional dynamic web application requires thinking in unconventional ways.

[SAMPLE REPORT -THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

4.10.1. Session Pollution

Applications should have logic controls to prevent them from accepting spurious requests. Such requests allow attackers to circumvent business logic or processes by predicting, finding, or manipulating parameters to make the application assume that a process or task did or did not occur.

The application only processed business logic streams for the same user in a sequential step order and there were no successful attempts to bypass or skip steps.

4.10.2. Request throttling

The application has been tested for a number of vulnerabilities that can be fixed by limiting the number of requests. Normally, a vulnerability may arise when an action, such as reusing a voucher, is allowed. However, such functionality not found in the respective application.

4.10.3. File Upload

Developers should always place restrictions on file uploads. For example, files should be limited to a maximum size, only certain types of files such as PDF-JPG should be allowed, and the service should check for malicious code.

4.10.3.1 Large Files

The application should limit the allowed file size based on the type, content requirements, and logic behind file usage. If an attacker sends a large file to the application, it may lead to degradation of application performance due to file processing time. On the other hand, it could lead to an unusable application due to storage limitations.

The application indicated that the maximum file size is 20972 KB. A 25 MB file was created and an upload was attempted.

```
fallocate -l 25M largeFile.png
```

The application gave the following error message:

The application did not accept large files.

4.10.3.2 Unexpected File Types

The application must handle not only the file contents safely, but also the file metadata. Metadata, such as the file path or file name, is usually provided by content transfer encryption, i.e., 'HTTP multipart encoding'. An attacker could use this data to overwrite a critical file within the application or to store the file in a particular location.

The application did not specify which file types were accepted. A php file was created:

```
echo "<? php TESTFILE (); ?>" > TESTFILE.php
```

This file has been uploaded:

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT IN DETAIL INCLUDING REQUEST - RESPONSE AND PROOF]

The file was refused:

```
Date: Tue , 31 Feb 1988 07:47:27 CMT [...]
Error 500: CWSRV0295E: Error reported: 500
```

No problems were found with the file upload functionality.

4.10.3.3 Upload of Malicious Files

The EICAR sequence was used to verify that uploaded files are scanned for viruses and malware,:

```
X5O!P%@AP [4\ PZX54(P^)7CC)7}$EICAR -STANDARD -ANTIVIRUS -TEST - FILE!$H+H*
```

When an application or script contains the EICAR sequence, it must activate the antivirus or malware software before saving the file.

4.10.4. File Download

4.11. Client-side

Browsers can store application data for maintaining browser history and for caching purposes. Caching is used to improve browser performance; content that has already been requested does not have to be reloaded for each new request. This can also improve usability by allowing users to re-use data entered onto a form when returning to the page. This has the downside of allowing an attacker with access to a user's browser to access potentially sensitive information, as it is stored locally in the browser.

4.11.1. Caching

The server used the following response headers to instruct the browser not to cache data for each page containing sensitive information:

```
Cache -Control: no -cache , no -store , must -revalidate.
```

4.11.2. Local Browser Storage

Local browser storage in the form of standard cookies has been found to be in use. However, no instances of local storage of sensitive information were found.

4.11.3. Sensitive personal information.

Sensitive information should never be sent from the application to the server using query string parameters. This applies to any HTTP method. By default, the values of these parameters are stored in the browser history and server log files. In addition, this information can be seen in the URL when "looking over the shoulder".

The assessment did not detect that sensitive information is being sent.

5. OWASP TOP 10 2021

[SAMPLE REPORT - THESE CHAPTERS ARE INCLUDED IN THE ACTUAL REPORT WITH ADDED INFORMATION ABOUT THE ACTUAL VULNERABILITIES MATCHED TO THE CORRECT CHAPTER!]

A01 Broken Access Control

Description

Access control enforces policies to ensure that users cannot act outside of intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits. Common access control vulnerabilities include:

- Violation of the principle of least privilege or denial by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone.
- Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool modifying API requests.
- Permitting viewing or editing someone else's account, by providing its unique identifier (insecure direct object references)
- Accessing API with missing access controls for POST, PUT and DELETE.
- Elevation of privilege. Acting as a user without being logged in or acting as an admin when logged in as a user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.
- CORS misconfiguration allows API access from unauthorized/untrusted origins.
- Force browsing to authenticate pages as an unauthenticated user or to privileged pages as a standard user.

Prevention

Access control is only effective in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

- Except for public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.
- Model access controls should enforce record ownership rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g., .git) and backup files are not present within web roots.

- Log access control failures, alert admins when appropriate (e.g., repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- Stateful session identifiers should be invalidated on the server after logout. Stateless JWT tokens should rather be short-lived so that the window of opportunity for an attacker is minimized. For longer lived JWTs it's highly recommended to follow the OAuth standards to revoke access.

Developers and QA staff should include functional access control unit and integration tests.

A02 Cryptographic Failures

Description

Cryptographic failures include failures to meet the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, mainly if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations, e.g., financial data protection such as PCI Data Security Standard (PCI DSS). For all such data the following security considerations apply:

- Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, FTP also using TLS upgrades like STARTTLS. External internet traffic is hazardous. Verify all internal traffic, e.g., between load balancers, web servers, or back-end systems.
- Are any old or weak cryptographic algorithms or protocols used either by default or in older code?
- Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing? Are crypto keys checked into source code repositories?
- Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing?
- Is the received server certificate and the trust chain properly validated?
- Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate?
- Are passwords being used as cryptographic keys in absence of a password base key derivation function?
- Is randomness used for cryptographic purposes that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability?
- Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are needed?
- Are deprecated cryptographic padding methods such as PKCS number 1 v1.5 in use?
- Are cryptographic error messages or side channel information exploitable, for example in the form of padding oracle attacks?

See ASVS Crypto (V7), Data Protection (V9), and SSL/TLS (V10)

Prevention

Do the following, at a minimum, and consult references:

- Classify data as processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
- Don't unnecessarily store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
- Encrypt all sensitive data at rest.
- Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; using proper key management.
- Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS)
- Disable caching for response that contain sensitive data.
- Apply required security controls as per the data classification.
- Do not use legacy protocols such as FTP and SMTP for transporting sensitive data.
- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2.
- Initialization vectors must be chosen according to the mode of operation. For many modes, this means using a CSPRNG (cryptographically secure pseudo random number generator). For modes that require a nonce, the initialization vector (IV) does not need a CSPRNG. In all cases, the IV should never be used twice for a fixed key.
- Always use authenticated encryption instead of just encryption.
- Keys should be generated cryptographically randomly and stored in memory as byte arrays. If a password is used, it must be converted to a key via an appropriate password base key derivation function.
- Ensure that cryptographic randomness is used where appropriate and that it has not been seeded in a predictable way or with low entropy. Most modern APIs do not require the developer to seed the CSPRNG to achieve security.
- Avoid deprecated cryptographic functions and padding schemes, such as MD5, SHA1, PKCS number 1 v1.5 .
- Independently verify the efficacy of configuration and settings.

A03 Injection

Description

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection.The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections. Automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs is strongly encouraged. Organizations can include static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline to identify introduced injection flaws before production deployment.

Prevention

Preventing injection requires keeping data separate from commands and queries:

- The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs).
- Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
- Note: SQL structures such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

A04 Insecure Design

Description

Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top 10 risk categories. There is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, as they have different root causes and remediation. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as by definition, security controls were never created to defend against all attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required.

Requirements and Resource Management

Collect and negotiate the business requirements for an application with the business, including the protection requirements concerning confidentiality, integrity, availability, and authenticity of all data assets and the expected business logic. Take into account how exposed your application will be and if you need segregation of tenants (in addition to access control). Compile the technical requirements, including functional and non-functional security requirements. Plan and negotiate the budget covering all design, build, testing, and operation, including security activities.

Secure Design

Secure design is a culture and methodology that constantly evaluates threats and ensures that code is robustly designed and tested to prevent known attack methods. Threat modeling should be integrated into refinement sessions (or similar activities); look for changes in data flows and access control or other security controls. In the user story development determine the correct flow and failure states, ensure they are well understood and agreed upon by responsible and impacted parties. Analyze assumptions and conditions for expected and failure flows, ensure they are still accurate and desirable. Determine how to validate the assumptions and enforce conditions needed for proper behaviors. Ensure the results are documented in the user story. Learn from mistakes and offer positive incentives to promote improvements. Secure design is neither an add-on nor a tool that you can add to software.

Secure Development Lifecycle

Secure software requires a secure development lifecycle, some form of secure design pattern, paved road methodology, secured component library, tooling, and threat modeling. Reach out to your security specialists at the beginning of a software project as well as throughout project and maintenance of your software. Consider leveraging the OWASP Software Assurance Maturity Model (SAMM) to help structure your secure software development efforts.

fPrevention

- Establish and use a secure development lifecycle with AppSec professionals to help evaluate and design security and privacy-related controls
- Establish and use a library of secure design patterns or paved road ready-to-use components
- Use threat modeling for critical authentication, access control, business logic, and key flows
- Integrate security language and controls into user stories
- Integrate plausibility checks at each tier of your application (from frontend to backend)
- Write unit and integration tests to validate that all critical flows are resistant to the threat model. Compile use-cases and misuse-cases for each tier of your application.
- Segregate tier layers on the system and network layers depending on the exposure and protection needs
- Segregate tenants robustly by design throughout all tiers
- Limit resource consumption by user or service

A05 Security Misconfiguration

Description

The application might be vulnerable if the application is or has:

- Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
- Unnecessary features enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords enabled and unchanged.
- Error handling revealing stack traces or other overly informative error messages to users.
- For upgraded systems, the latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
- The server does not send security headers or directives, or they are not set to secure values.
- The software is out of date or vulnerable (see A06:2021-Vulnerable and Outdated Components).
- Without a concerted, repeatable application security configuration process, systems are at a higher risk.

Prevention

Secure installation processes should be implemented, including:

- A repeatable hardening process makes it fast and easy to deploy another environment that is appropriately locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to set up a new secure environment.
- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
- A task to review and update the configurations appropriate to all security notes, updates, and patches as part of the patch management process (see A06:2021-Vulnerable and Outdated Components). Review cloud storage permissions (e.g., S3 bucket permissions).
- A segmented application architecture provides effective and secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).
- Sending security directives to clients, e.g., Security Headers.
- An automated process to verify the effectiveness of the configurations and settings in all environments.

A06 Vulnerable and Outdated Components

Description

You are likely vulnerable:

- If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.
- If the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
- If you do not scan for vulnerabilities regularly in addition to subscribing to security bulletins related to the components you use.

- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure to fixed vulnerabilities.
- If software developers do not test the compatibility of updated, upgraded, or patched libraries.
- If you do not secure the components' configurations (see A05:2021-Security Misconfiguration).

Prevention

There should be a patch management process in place to:

- Remove unused dependencies, unnecessary features, components, files, and documentation.
- Continuously take inventory of both client-side and server-side component versions (e.g., frameworks, libraries) and their dependencies using tools like versions, OWASP Dependency Check, retire.js, etc.
- Continuously monitor sources like Common Vulnerability and Exposures (CVE) and National Vulnerability Database (NVD) for vulnerabilities in the components.
- Use software composition analysis tools to automate the process.
- Subscribe to email alerts for security vulnerabilities related to components you use.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component (See A08:2021-Software and Data Integrity Failures).
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

Every organization must ensure an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.

A07 Identification and Authentication Failures

Description

Confirmation of the user's identity, authentication, and session management is critical to defending against authentication-related attacks. There may be authentication weaknesses if the application:

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords data stores (see A02:2021-Cryptographic Failures).
- Has missing or ineffective multi-factor authentication.
- Exposes session identifier in the URL.

- Reuse session identifier after successful login.
- Does not correctly invalidate Session IDs. User sessions or authentication tokens (mainly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

Prevention

- Where possible, implement multi-factor authentication to prevent automated credential stuffing, brute force, and stolen credential reuse attacks.
- Do not ship or deploy with any default credentials, particularly for admin users.
- Implement weak password checks, such as testing new or changed passwords against the top 10,000 worst passwords list.
- Align password length, complexity, and rotation policies with National Institute of Standards and Technology (NIST) 800-63b's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence-based password policies.
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts, but be careful not to create a denial of service scenario. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session identifier should not be in the URL, be securely stored, and invalidated after logout, idle, and absolute timeouts.

A08 Software and Data Integrity Failures

Description

Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. An example is when an application relies on plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline can introduce the potential for unauthorized access, malicious code, or system compromise. Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.

Prevention

- Use digital signatures or similar mechanisms to verify the software or data is from the expected source and has not been altered.
- Ensure libraries and dependencies, such as npm or Maven, are consuming trusted repositories. If you have a higher risk profile, consider hosting a vetted, internal known-good repository
- Ensure that a software supply chain security tool, such as OWASP Dependency Check or OWASP CycloneDX, is used to verify that components do not contain known vulnerabilities

- Ensure that there is a review process for code and configuration changes to minimize the chance that malicious code or configuration could be introduced into your software pipeline.
- Ensure that your CI/CD pipeline has proper segregation, configuration, and access control to ensure the integrity of the code flowing through the build and deploy processes.
- Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients without some form of integrity check or digital signature to detect tampering or replay of the serialized data

A09 Security Logging and Monitoring Failures

Description

Returning to the OWASP Top 10 2021, this category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected. Insufficient logging, detection, monitoring, and active response occurs any time:

- Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally.
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
  *The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.
- You are vulnerable to information leakage by making logging and alerting events visible to a user or an attacker (see A01:2021-Broken Access Control).

Prevention

Developers should implement some or all the following controls, depending on the risk of the application:

- Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.
- Ensure that logs are generated in a format that log management solutions can easily consume.
- Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.
- Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- DevSecOps teams should establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.
- Establish or adopt an incident response and recovery plan, such as National Institute of Standards and Technology (NIST) 800-61r2 or later.

There are commercial and open-source application protection frameworks such as the OWASP ModSecurity Core Rule Set, and open-source log correlation software, such as the Elasticsearch, Logstash, Kibana (ELK) stack, that feature custom dashboards and alerting.

A10 Server Side Request Forgery (SSRF)

Description

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).

As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

Prevention

Developers can prevent SSRF by implementing some or all the following defense in depth controls:

From the Network Layer
*Segment remote resource access functionality in separate networks to reduce the impact of SSRF
*Enforce “deny by default” firewall policies or network access control rules to block all but essential intranet traffic.
Hints:

- Establish ownership and a lifecycle for firewall rules based on applications.
- Log all accepted and blocked network flows on firewalls (see A09:2021-Security Logging and Monitoring Failures).

From the Application Layer

- Sanitize and validate all client-supplied input data
- Enforce the URL schema, port, and destination with a positive allow list
- Do not send raw responses to clients
- Disable HTTP redirections
- Be aware of the URL consistency to avoid attacks such as DNS rebinding and “time of check, time of use” (TOCTOU) race conditions
- Do not mitigate SSRF via the use of a deny list or regular expression. Attackers have payload lists, tools, and skills to bypass deny lists.

Additional Measures to Consider:

- Don't deploy other security relevant services on front systems (e.g. OpenID). Control local traffic on these systems (e.g. localhost)
- For frontends with dedicated and manageable user groups use network encryption (e.g. VPNs) on independent systems to consider very high protection needs

6. Findings

6.1. Summary

6.1.1. Vulnerability Summary

Vulnerability	Severity	CVSS 3.1
F-2021-0937 - Cross-site scripting (stored)	High	8.2
F-2021-0938 - Cross-site scripting (reflected)	High	7.6
F-2021-0939 - Form does not contain an anti-CSRF token	High	7.1
F-2021-0933 - CSP: Inline scripts can be inserted	Medium	5.3
F-2021-0934 - Client-side HTTP parameter pollution (reflected)	Low	3.7
F-2021-0941 - Strict transport security not enforced	Low	3.3

6.1.2. Observations Summary

Observation
F-2021-0935 - Cross-domain Referer leakage

6.2. Vulnerability Details

F-2021-0937 - Cross-site scripting (stored)

Stored cross-site scripting vulnerabilities arise when user input is stored and later embedded into the application's responses in an unsafe way.

Classification

Severity: High

Impact: 4/5

Likelihood: 3/5

CVSS 3.1: 8.2 (CVSS:3.1/AV:N/AC:L/PR:L/UI:R/S:C/C:L/I:H/A:L)

CWEs:

- CWE-79 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-80 | Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
- CWE-116 | Improper Encoding or Escaping of Output
- CWE-159 | Improper Handling of Invalid Use of Special Elements

Recommendations

Background Information

Stored cross-site scripting vulnerabilities arise when user input is stored and later embedded into the application's responses in an unsafe way. An attacker can use the vulnerability to inject malicious JavaScript code into the application, which will execute within the browser of any user who views the relevant application content.

The attacker-supplied code can perform a wide variety of actions, such as stealing victims' session tokens or login credentials, performing arbitrary actions on their behalf, and logging their keystrokes.

Methods for introducing malicious content include any function where request parameters or headers are processed and stored by the application, and any out-of-band channel whereby data can be introduced into the application's processing space (for example, email messages sent over SMTP that are ultimately rendered within a web mail application).

Stored cross-site scripting flaws are typically more serious than reflected vulnerabilities because they do not require a separate delivery mechanism in order to reach target users, and are not hindered by web browsers' XSS filters. Depending on the affected page, ordinary users may be exploited during normal use of the application. In some situations this can be used to create web application worms that spread exponentially and ultimately exploit all active users.

Note that automated detection of stored cross-site scripting vulnerabilities cannot reliably determine whether attacks that are persisted within the application can be accessed by any other user, only by authenticated users, or only by the attacker themselves. You should review the functionality in which the vulnerability appears to determine whether the application's behavior can feasibly be used to compromise other application users.

Recommendations

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 <br>it is expected to contain. For example, personal names should consist of alphabetical <br>and a small range of typographical characters, and be relatively short; a year of birth <br>should consist of exactly four numerals; email addresses should match a well-defined <br>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 <br>application responses. All HTML metacharacters, including < > " ' and =, should be <br>replaced with the corresponding HTML entities (< > 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.

External Url: https://portswigger.net/web-security/cross-site-scripting

F-2021-0938 - Cross-site scripting (reflected)

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.

Classification

Severity: High

Impact: 4/5

Likelihood: 3/5

CVSS 3.1: 7.6 (CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:L/I:H/A:L)

CWEs:

- CWE-79 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-80 | Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
- CWE-116 | Improper Encoding or Escaping of Output
- CWE-159 | Improper Handling of Invalid Use of Special Elements

Recommendations

Background Information

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.

Recommendations

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 <br>it is expected to contain. For example, personal names should consist of alphabetical <br>and a small range of typographical characters, and be relatively short; a year of birth <br>should consist of exactly four numerals; email addresses should match a well-defined <br>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 <br>application responses. All HTML metacharacters, including < > " ' and =, should be <br>replaced with the corresponding HTML entities (< > 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.

External Url: https://portswigger.net/web-security/cross-site-scripting

F-2021-0939 - Form does not contain an anti-CSRF token

Cross-site Request Forgery (CSRF) is an attack which forces an end user to execute unwanted actions on a web application to which he/she is currently authenticated.

Classification

Severity: High

Impact: 4/5

Likelihood: 2/5

CVSS 3.1: 7.1 (CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:H/I:H/A:L)

Recommendations

Background Information

Cross-site Request Forgery (CSRF) is an attack which forces an end user to execute unwanted actions on a web application to which he/she is currently authenticated. With a little help of social engineering (like sending a link via email / chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and may allow an attacker to perform an account hijack. If the targeted end user is the administrator account, this can compromise the entire web application.

Recommendations

The application should implement anti-CSRF tokens into all requests that perform actions which change the application state or which add/modify/delete content. An anti-CSRF token should be a long randomly generated value unique to each user so that attackers cannot easily brute-force it.

It is important that anti-CSRF tokens are validated when user requests are handled by the application. The application should both verify that the token exists in the request, and also check that it matches the user's current token. If either of these checks fails, the application should reject the request.

F-2021-0933 - CSP: Inline scripts can be inserted

CSP: Inline scripts can be inserted

Assets:

- Cyver Demo App [https://cyver-demo-app.io]

Classification

Severity: Medium

Impact: 2/5

Likelihood: 3/5

CVSS 3.1: 5.3 (CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L)

F-2021-0934 - Client-side HTTP parameter pollution (reflected)

Client-side HTTP parameter pollution (HPP) vulnerabilities arise when an application embeds user input in URLs in an unsafe manner.

Assets:

- Cyver Demo App [https://cyver-demo-app.io]

Classification

Severity: Low

Impact: 2/5

Likelihood: 3/5

CVSS 3.1: 3.7 (CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N)

CWEs:

- CWE-20 | Improper Input Validation
- CWE-233 | Improper Handling of Parameters

Recommendations

Background Information

Client-side HTTP parameter pollution (HPP) vulnerabilities arise when an application embeds user input in URLs in an unsafe manner. An attacker can use this vulnerability to construct a URL that, if visited by another application user, will modify URLs within the response by inserting additional query string parameters and sometimes overriding existing ones. This may result in links and forms having unexpected side effects. For example, it may be possible to modify an invitation form using HPP so that the invitation is delivered to an unexpected recipient.

The security impact of this issue depends largely on the nature of the application functionality. Even if it has no direct impact on its own, an attacker may use it in conjunction with other vulnerabilities to escalate their overall severity.

Recommendations

Ensure that user input is URL-encoded before it is embedded in a URL.

External Url: https://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf

F-2021-0941 - Strict transport security not enforced

The application fails to prevent users from connecting to it over unencrypted connections.

Classification

Severity: Low

Impact: 1/5

Likelihood: 3/5

CVSS 3.1: 3.3 (CVSS:3.1/AV:N/AC:H/PR:H/UI:N/S:U/C:L/I:N/A:L)

CWEs:

- CWE-523 | Unprotected Transport of Credentials

Recommendations

Background Information

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.

Recommendations

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.

External Url: https://developer.mozilla.org/en-US/docs/Web/Security/HTTP_strict_transport_security

6.3. Observations Details

F-2021-0935 - Cross-domain Referer leakage

When a web browser makes a request for a resource, it typically adds an HTTP header, called the "Referer" header, indicating the URL of the resource from which the request originated.

Classification

Likelihood: 3/5

CWEs:

- CWE-200 | Exposure of Sensitive Information to an Unauthorized Actor

Recommendations

Background Information

When a web browser makes a request for a resource, it typically adds an HTTP header, called the "Referer" header, indicating the URL of the resource from which the request originated. This occurs in numerous situations, for example when a web page loads an image or script, or when a user clicks on a link or submits a form.

If the resource being requested resides on a different domain, then the Referer header is still generally included in the cross-domain request. If the originating URL contains any sensitive information within its query string, such as a session token, then this information will be transmitted to the other domain. If the other domain is not fully trusted by the application, then this may lead to a security compromise.

You should review the contents of the information being transmitted to other domains, and also determine whether those domains are fully trusted by the originating application.

Today's browsers may withhold the Referer header in some situations (for example, when loading a non-HTTPS resource from a page that was loaded over HTTPS, or when a Refresh directive is issued), but this behavior should not be relied upon to protect the originating URL from disclosure.

Note also that if users can author content within the application then an attacker may be able to inject links referring to a domain they control in order to capture data from URLs used within the application.

Recommendations

Applications should never transmit any sensitive information within the URL query string. In addition to being leaked in the Referer header, such information may be logged in various locations and may be visible on-screen to untrusted parties. If placing sensitive information in the URL is unavoidable, consider using the Referer-Policy HTTP header to reduce the chance of it being disclosed to third parties.

External Url: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy

7. Test Setup

7.1. SOURCE IP-ADDRESS(ES)

`Source IP addres EthicalHacker 1 : 21x.12x.15x.16x`

7.2. HARDWARE AND OS

```
Hardware and OS used for this test: System 1 : MacBook Pro (16-inch, 2019) 2,4 GHz
8-Core Intel Core i9 32 GB 2667 MHz DDR4 AMD Radeon Pro 5500M 8 GB macOS Big Sur
versie 11.4
```

`System 2: Lenovo`

7.3. TESTTOOLS AND VERSIONNUMBERS

```
Nessus Professional version 8 8.15.0 (#271) DARWIN Burp Suite Professional
2021.6.2 NMAP version 7.91 Postman 8.7.0 SSL scan
```

7.4 TESTDATA

Original test data is available to the client up to 2 months after the report. By this we mean the Burp Pro project - state file., Nessus Pro file, notes, screen captures or output data from, for example, NMAP or SQLMAP.