Web App Penetration Test
Sample App

Prepared for Acme, Inc

John Doe
jdoe@example.com

January 1, 2026

Adversis, LLC

PO Box 2953
Kalispell, MT 59901
+1 (877) 353-1337
hello@adversis.io

Contents

Executive Summary

App	ACME App
Type	Web Application Penetration Test
Scope	acmetest.example.com
Environment	Staging
Dates	January 1 to January 14, 2025
Outcome	Robust Controls

Overview

Adversis conducted a black-box assessment of Acme Inc's ACME Application above and beyond OWASP ASVS Level 1 methodology.

Testing evaluated the web application and REST API powering ACME's App solution using SDK-based and direct REST approaches. This assessment covered all major features, including create, read, update, delete (CRUD) operations for user management, contact management, and reporting functionality.

Aspects of regulatory compliance pertaining to key data elements were also considered.

Key Findings

The ACME application uses a robust web framework and configuration, inherently providing strong defenses against common vulnerabilities. Specifically, the authentication and authorization framework is industry standard and well-designed with secure features and defaults. Unlike using raw SQL queries, the app also uses safe, language-integrated database queries in the code to prevent vulnerabilities that can lead to data loss.

Critically, a reliance on insufficient built-in input filters and missing output encoding capabilities introduces stored cross-site scripting vulnerabilities, which can lead to account compromise and eventual data leakage.

Additionally, the absence of role authorization checks during report execution enables some users to access information they should not be able to, although highly sensitive information remains restricted through appropriate database checks.

All identified findings are resolvable through configuration and code modifications without architectural redesign. The most critical path to address involves implementing output encoding and restricting administrative access to diagnostic tools.

Positive Controls

The ACME application demonstrates strong security measures and robust configuration, enhancing its resilience against common cyber threats. These include:

- A robust web framework and secure configuration
- Secure account authentication library usage and strong data handling measures

In particular, the team identified several items worth championing.

- Strong authentication: The OWIN authentication middleware automatically implements essential security features, which help mitigate vulnerabilities in account takeovers. In addition, the application enforces an automatic session timeout with a judiciously set duration, minimizing the risk of unauthorized access during periods of inactivity.

- Secure File Handling: Files cannot be uploaded or made accessible without proper authentication. Additionally, files are neither hosted nor executed on the application server; instead, Azure services and unique identifiers are used, providing an extra layer of security.

- Validation and Encryption: Both server and client-side validations are rigorously applied, including measures against Cross-Site Request Forgery (CSRF) and encryption of ViewState data. These controls are essential for maintaining the integrity and confidentiality of user interactions.

- SQL Injection Protection: Using Linq to Entities Framework for database interactions effectively prevents SQL injection, a prevalent threat that can lead to significant data loss.
- Effective Authorization Checks: The application exhibits a well-implemented system for user roles and authorization verification, ensuring access rights are strictly enforced according to user privileges.

Overall, the ACME web application's security posture is commendable. It reflects a thoughtful and proactive approach to protecting data and user interactions against a range of threats.

Strategic Recommendations

To build upon this strong foundation, several approaches should be considered.

1. **Strengthen Output Handling to Prevent Account Compromise**
   Ensure all user-provided information is encoded before displaying it to protect against Cross-Site Scripting (XSS) attacks. This practice ensures that any special characters are converted into HTML entities, preventing them from being interpreted as code.

   Implementing HTML encoding systematically across the application will fortify your defenses against one of the most common web security threats and prevent account compromise and data exfiltration. Fortunately, the .NET HtmlEncode method can be used to accomplish this easily.

2. **Restrict Access to Diagnostic Tools**
   Modify the web.config file to limit the error logging system ELMAH to localhost only or further limit access to restricted admin roles to prevent admin privilege escalation.

3. **Enforce Authorization on Report Generation**
   Authorization controls for data and report execution should be bolstered. Given the sensitivity of the information in reports, implementing additional authorization requirements will help ensure that only qualified users can access or generate reports.

Conclusion

Adversis has highlighted the ACME application's strong security and effective protective measures through this penetration test against a variety of threats.

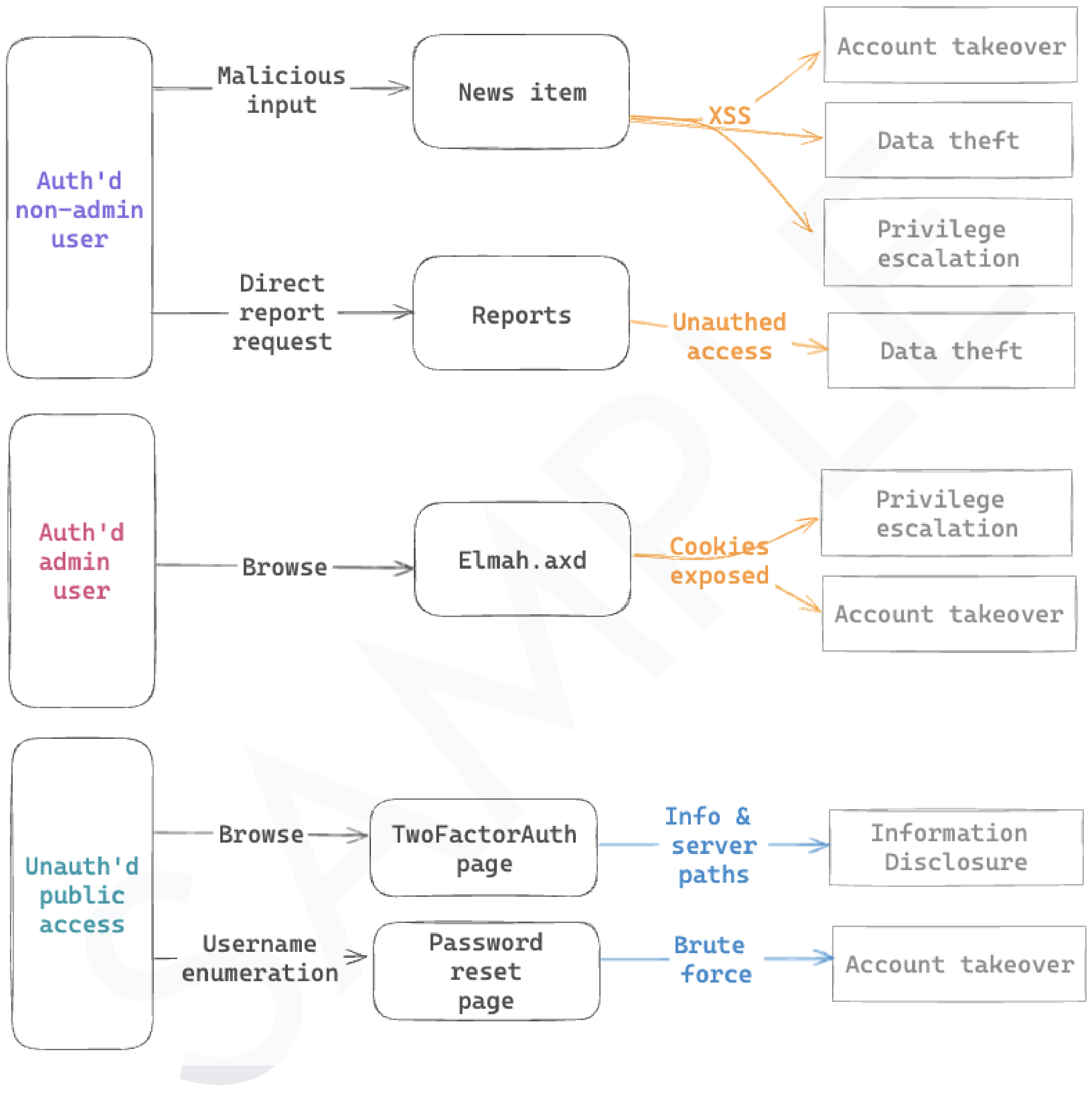
By utilizing secure coding practices, robust session management, and advanced security mechanisms, the application maintains a high level of protection for both data and user interactions.

To build upon this strong foundation, we recommend implementing HTML encoding for all outputs and enhancing authorization protocols for report access.

These enhancements will secure the application against emerging threats and ensure it remains resilient in a dynamic security environment.

Attack Summary Diagram

Following is a basic visual description of issues identified in the ACME application.



Findings Overview

ID	Description	Risk
1	**Stored Cross-Site Scripting (XSS) Vulnerability in Contact Form** Filter bypass and a lack of output encoding allow attackers to introduce arbitrary code that affects a user's browsing, typically leading to account takeover.	Moderate
2	**Error Logging Modules and Handlers (ELMAH) Exposed to Admins** All admin users can view sensitive authentication cookies for other users, which could allow privilege escalation and unauthorized data access.	Moderate
3	**Unauthorized Report Execution by Low-Privilege Users** Low-privileged users can run reports and obtain data from reports typically not accessible to them by specifying the report to run.	Moderate
4	**Username Enumeration Vulnerability in Forgot Password Flow** Login messages vary their responses, allowing an attacker to determine usernames to facilitate account takeover and brute force attempts.	Low
5	**Missing Content-Security-Policy Security Header** Missing security headers is a defense in depth configuration to decrease the impact of cross-site scripting and other client-side vulnerabilities.	Low
6	**Information Disclosure in Detailed Error Messages** An unhandled error on an unauthenticated login page returns sensitive information about the application, including software version info, server paths, and file names.	Informational

Findings Details

1. Stored XSS via Unicode Normalization Bypass

Vulnerability	Contact form stores and renders unsanitized user input after Unicode normalization bypasses ASP.NET ValidateRequest
Exploited via	Dotless 'ı' (U+0131) input normalizes to 'i' post-validation, constructs <iframe> tag
Immediate Risk	Admin users reviewing Contact submissions will execute attacker payloads
Severity	**Moderate**. Authentication required, affects all users.
Attack Surface	2 confirmed endpoints, pattern likely exists elsewhere
Fix	Context-aware output encoding (HtmlEncode) and CSP deployment

Description

The Contact form performs Unicode normalization after ASP.NET's ValidateRequest filter executes. When users submit dotless 'ı' (U+0131) characters, they bypass the filter but normalize to standard 'i' (U+0069) post-validation, reconstructing filtered HTML tags like `<iframe>`. The application then renders Contact submissions using `InnerHtml` without output encoding, despite using `HtmlEncode` correctly on the home page.

Impact

Administrative staff view Contact submissions daily as part of the typical workflow. Malicious payloads could execute in the context of authenticated sessions with elevated privileges, including access to customer PII, payment information, and administrative functions. The stored nature means a single successful submission affects multiple staff members over time. The current lack of CSP means payloads can load external resources and exfiltrate data to attacker-controlled domains.

Context

Why this exists
The application relies on ASP.NET's ValidateRequest filter as its primary XSS defense, but this only validates input before Unicode normalization occurs. The application performs Unicode normalization after validation, converting dotless 'ı' (U+0131) to standard 'i' (U+0069), which reconstructs filtered HTML tags post-validation.

Architecture gap
Defense-in-depth principle was violated with a single control (ValidateRequest) at the wrong layer. No output encoding was implemented.

Location

This vulnerability exists in at least 2 confirmed locations

- `https://acmetest.example.com/Contact.aspx?Key=16`
- `https://acmetest.example.com/User.aspx?Key=127`

Remediation Approach

Primary Fix

- Use context-sensitive encoding, such as the HttpUtility.HtmlEncode to encode all user-supplied input before rendering it on any page, ensuring that any malicious scripts are neutralized. The XSS doesn't render on the home page since HtmlEncode is used. Microsoft also states not to rely on ASP.NET's ValidateRequest filter.[1]

Secondary Fix

- Implement a Content Security Policy (CSP) to reduce the severity of any remaining XSS vulnerabilities, including restricting the loading of remote, attacker-supplied resources. (See *Missing Content-Security-Policy Header* finding).

Acceptance Criteria

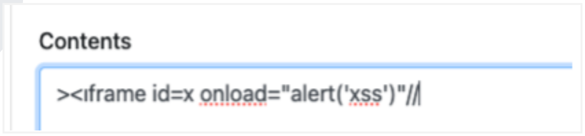
- All user input from Contact form is HTML-encoded before rendering
- ValidateRequest bypass via Unicode normalization is no longer possible
- CSP header deployed to limit XSS impact
- Pattern verified fixed across User.aspx and other input forms

Details

Within the Contact page, create a new item. Enter the following text:

```
><ıframe id=x onload="alert('xss')"//
```

Note that a dotless 'ı' character is used in place of the normal "i" character. Typically, the ValidateRequest filter will prevent this input, but the Unicode letter bypasses the filter. Once the Contact item is saved, it is converted to a standard "i", spelling "iframe". The code executes once the Contact item is viewed.


Contents

><ıframe id=x onload="alert('xss')"//

A malicious payload about to be saved in the Contact item

The encoded payload will execute if stored and rendered without proper sanitization. As identified in the code, the Contact entry is simply appended to the HTML output without being safely encoded.

[1] https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff647397(v=pandp.10)

2. Error Logging Modules and Handlers (ELMAH) Exposed to All Admin Roles

Vulnerability	The ELMAH web interface exposes session cookies in error logs to all administrator roles, enabling horizontal and vertical privilege escalation
Exploited via	Access elmah.axd with any admin credentials, view error details, extract .AspNet.ApplicationCookie from other users' requests
Immediate Risk	Application Admins can escalate to Super Admin by capturing session cookies from error logs
Severity	**Medium.** Requires admin authentication, affects all admin tiers
Attack Surface	Single endpoint (/elmah.axd), impacts all users who trigger errors
Fix	Restrict ELMAH access to Super Admin role only via custom controller with granular authorization

Description

ELMAH is configured with `allowRemoteAccess="true"`, and the web.config authorization allows all users in the admin role to access all resources without distinguishing between privilege tiers. The error logging interface displays complete request details, including the `AspNet.ApplicationCookie` values for every logged exception. The application has multiple admin roles (Application Admin, Super Admin) with different privilege levels, but ELMAH's authorization treats them identically.

Impact

Application Admins regularly access diagnostic tools as part of troubleshooting workflows. When Super Admins trigger application errors during their normal operations, their session cookies become visible to any admin viewing `elmah.axd`, enabling session hijacking without credential theft. Compromised Super Admin sessions provide unrestricted access to user management, system configuration, and all administrative functions that Application Admins normally cannot access.

Context

Why this exists
ELMAH was deployed with its default permissive configuration (`allowRemoteAccess="true"`) and protected only with role-based authentication checking for "admin" membership. The web.config authorization was implemented without considering that admin roles have different privilege levels within the application, treating all administrators as equally trusted.

Architecture gap

- Principle of least privilege violated - diagnostic tools with sensitive data exposure should be restricted to highest privilege tier only.

- Defense-in-depth missing - no secondary controls prevent session token leakage in logs.

Location

This vulnerability exists at

- https://acmetest.example.com/elmah.axd
- Configuration: `web.config` elmah section with `allowRemoteAccess="true"` and authorization allowing all admin roles

Resources

- http://beletsky.net/2011/03/integrating-elmah-to-aspnet-mvc-in.html

Remediation Approach

Primary Fix

- Remove the default ELMAH handler from web.config and implement a custom controller that restricts access to Super Admin role only.

```
// Remove from web.config
<location path="elmah.axd">
  <system.web>
    <authorization>
      <allow roles="admin" />
      <deny users="*" />
    [...snip...]

// Add custom controller ElmahController.cs (and register route)
[Authorize(Roles = "SuperAdmin")]
public class ElmahController : Controller {
    public ActionResult Index() {
        return new ElmahResult();
```

Secondary Fix

- Consider restricting ELMAH access to localhost only or disabling it altogether.

```
<elmah>
  <security allowRemoteAccess="false" />
</elmah>
<location path="elmah.axd">
  <system.webServer>
    <security>
      <ipSecurity allowUnlisted="false">
         <add allowed="true" ipAddress="127.0.0.1" />
         <add allowed="true" ipAddress="::1" />
</ipSecurity> </security> </system.webServer> </location>
```

- Implement logging middleware that strips sensitive headers and cookies before logging exceptions.

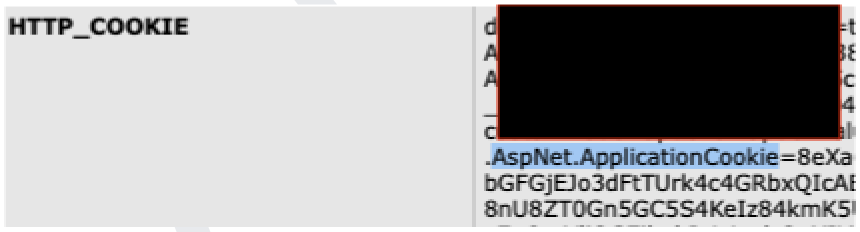
```
void ErrorLog_Filtering(object sender, ExceptionFilterEventArgs e) {
    var httpContext = e.Context as HttpContext;
    if (httpContext != null) {
        // Redact cookie values
        foreach (string cookie in httpContext.Request.Cookies) {
           if (cookie.StartsWith(".ASPXAUTH") || cookie.Contains("SessionId"))
{
              httpContext.Items[cookie] = "[REDACTED]";
```

Acceptance Criteria

- ELMAH access restricted to Super Admin role only
- Application Admin users receive 403 Forbidden when accessing **elmah.axd**
- Super Admin users can still access error logs with valid credentials
- Session cookies and authentication tokens are sanitized from error logs

Details

Authenticate to the application as an admin user and browsing to the /elmah.axd page to observer a list of errors logged by the application. After selecting an errors message, debug information for the user's request is returned, including the .AspNet.ApplicationCookie value, which allows anyone to impersonate that user's session.



HTTP_COOKIE	[redacted] .AspNet.ApplicationCookie=8eXa bGFGjEJo3dFtTUrk4c4GRbxQIcAE 8nU8ZT0Gn5GC5S4KeIz84kmK5

Session cookie value exposed in the elmah.axd log

3. Unauthorized Report Execution by Low-Privilege Users

Vulnerability	Low-privileged users can execute reports beyond their permissions by manipulating request parameters, bypassing UI-level access controls
Exploited via	Authenticate as low-privilege user, navigate to Reports. Intercept POST request, modify RunReport parameter to target unauthorized report ID, receive report export
Immediate Risk	Information disclosure of user PII to unauthorized roles
Severity	**Moderate**. Requires authenticated low-privilege account, affects reporting module
Attack Surface	Endpoint (/Reports/Reports.aspx), impacts reports lacking server-side authorization
Fix	Implement server-side authorization checks validating user permissions

Description

The reporting module enforces access controls at the UI layer only. While the interface restricts which reports are displayed based on user role, the server-side execution logic does not validate whether the requesting user has permission to run the specified report. By intercepting the POST request and modifying the `RunReport` parameter value, low-privileged users can execute arbitrary reports by specifying the target report ID directly.

Impact

Unauthorized users can export reports containing user PII, including email addresses and phone numbers. While the UI hides certain reports from low-privileged roles, the lack of server-side enforcement means any authenticated user can extract data from any report if they know or guess the report ID. This breaks the intended data segregation between privilege tiers.

Context

Why this exists
Authorization was implemented at the presentation layer only, assuming UI restrictions would prevent unauthorized access. The server trusts client-supplied report IDs without validating the user's permission to execute them.

Architecture gap

Broken access control—server-side authorization checks are missing for report execution. Defense relies on UI restrictions, which are trivially bypassed by manipulating requests.

Location

This vulnerability exists at:

- https://acmetest.example.com/Reports/Reports.aspx

- Server-side report execution handler accepting RunReport and RunName parameters

Resources

- https://owasp.org/Top10/A01_2021-Broken_Access_Control/

Remediation Approach

Primary Fix

Implement server-side authorization validation before executing any report. Verify the authenticated user's role has explicit permission for the requested report ID.

Secondary Fix

Validate reports against user permissions or roles, and add audit logging for all report execution attempts.

Acceptance Criteria

- Server-side authorization check validates user permission before report execution
- Unauthorized report requests return 403 Forbidden with no report data
- Audit log captures unauthorized access attempts with user ID and target report

Details

Log in with a low-privileged user, navigate to Reports, and select the "Log Analysis Export" report. Intercept the request and modify the report form values by specifying an unauthorized report ID:

```
# As a low-privileged user
POST /Reports/Reports.aspx HTTP/1.1
Host: acmetest.example.com
…[snip]…
Report=3
Name=LogAnalysisExport


# Response
HTTP/1.1 200 OK
Content-Type: application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
Content-Disposition: attachment; filename=LogAnalysisExport.xlsx
```

HTTP request and response from a user requesting an unauthorized report

The web application returns a user data export containing users' email addresses and phone numbers.

4. Username Enumeration Vulnerability in Forgot Password Flow

Vulnerability	Forgot password functionality returns distinct responses for valid vs. invalid usernames, enabling account enumeration
Exploited via	Submit invalid username and observe "user doesn't exist" message. Submit valid username and observe "email sent" message. Enumerate valid accounts by comparing responses
Immediate Risk	Attackers can compile a list of valid usernames for targeted password attacks or phishing campaigns
Severity	**Low.** No authentication required, Info disclosure
Attack Surface	A single endpoint (/Account/Forgot.aspx) exposes emails to enumeration
Fix	Implement uniform response messaging regardless of username validity

Description

The forgot password functionality reveals the validity of the username through differential responses. When a nonexistent username is submitted, the application returns an explicit error indicating that the user doesn't exist. When a valid username is submitted, the application confirms that an email has been sent. This behavioral difference allows unauthenticated attackers to systematically probe the endpoint and determine which usernames are registered in the system.

Impact

Attackers can harvest valid usernames without authentication. This enumerated list enables targeted attacks, including slow-and-low password brute forcing designed to evade lockout policies, credential stuffing using passwords from breached databases, and phishing campaigns targeting confirmed users. The application does not disclose associated email addresses, which limits the precision of spear-phishing attacks.

Context

Why this exists
The forgot password handler was implemented with user-friendly error messaging that prioritizes clarity over security, explicitly informing users when their username is not found rather than returning an ambiguous response.

Architecture gap
Information leakage through error handling—authentication-adjacent functions should not confirm or deny the existence of accounts to unauthenticated users.

Location

This vulnerability exists at

- https://acmetest.example.com/Account/Forgot.aspx
- Server-side handler returning differential responses based on username lookup results

Resources

- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/03-Identity_Management_Testing/04-Testing_for_Account_Enumeration_and_Guessable_User_Account

Remediation Approach

Primary Fix

Modify the forgot password response to return identical messaging regardless of whether the username exists in the system. For example, "If your username is recognized, a password reset email will be sent."

Secondary Fix

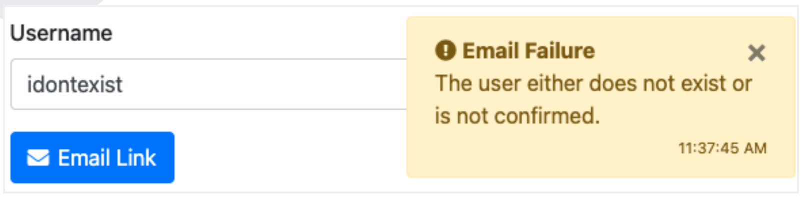
Implement rate limiting and monitoring to detect enumeration attempts, even with uniform messaging in place. For example, within middleware on the forgot password endpoint, limit responses to requests to 5 within a 5 minute period.

Acceptance Criteria

- Forgot password returns an identical response for valid and invalid usernames
- Response timing is consistent regardless of username validity (prevents timing-based enumeration)
- Rate limiting prevents rapid enumeration attempts
- Audit logging captures forgotten password attempts for anomaly detection

Details

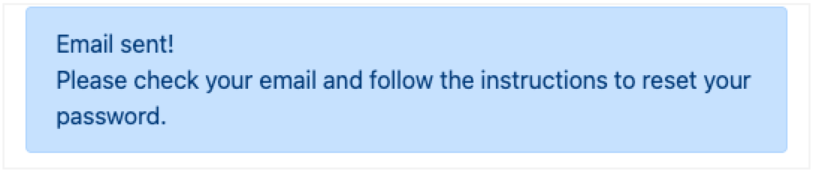
Access the Forgot Password Page. Enter a username that you know does not exist. Note the specific error message stating that the user doesn't exist.



Email Failure
The user either does not exist or is not confirmed.
11:37:45 AM

Application informing the user that the user does not exist

Enter a known valid username and observe that the response changes to indicate an email has been sent.


Email sent!
Please check your email and follow the instructions to reset your password.

Application informing the user that the provided username exists

The difference in error messages allows attackers to build a list of usernames that can be used in slow password attacks over time.

5. Missing Content-Security-Policy Security Header

Vulnerability	Application does not implement Content-Security-Policy header, removing a key defense-in-depth control against script injection attacks
Exploited via	Attacker identifies XSS vector and injects malicious script which browser executes without CSP restrictions leading to session hijacking or data exfil
Immediate Risk	XSS attacks face only browser-level mitigations, increasing impact of any injection vulnerabilities
Severity	Low. Defense-in-depth gap, affects entire application
Attack Surface	All application responses, global impact on client-side security posture
Fix	Implement Content-Security-Policy header restricting resource loading to trusted sources

Description

The application does not return a Content-Security-Policy (CSP) header in HTTP responses. CSP is a browser security mechanism that restricts which sources can load scripts, stylesheets, images, and other resources. Without CSP, browsers will execute any script injected into the page regardless of origin, providing no mitigation layer if XSS vulnerabilities exist elsewhere in the application.

Impact

The absence of CSP removes a defense-in-depth control against cross-site scripting. If an attacker discovers an XSS vector, malicious scripts execute without restriction—enabling session token theft, keylogging, DOM manipulation, and data exfiltration. CSP would otherwise block or limit unauthorized script execution even when injection succeeds. This gap increases the blast radius of any XSS vulnerability from potentially contained to fully exploitable.

Context

Why this exists
CSP was not included in the original security header configuration. Implementation requires auditing all script sources, inline scripts, and third-party resources, which may have been deferred during development.

Architecture gap
Defense-in-depth missing—application relies solely on input validation and output encoding to prevent XSS, with no browser-enforced fallback.

Location

This vulnerability exists at

- All HTTP responses from https://acmetest.example.com
- Web server configuration (web.config or Startup.cs) is missing CSP header definition

Resources

- https://content-security-policy.com/
- https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html
- https://report-uri.com/home/analyse

Remediation Approach

Primary Fix

Implement a Content-Security-Policy header in web.config that restricts loading of resources to trusted sources. Start with a restrictive policy and expand as needed.

```
<!-- web.config -->
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="Content-Security-Policy"
         value="default-src 'self';
             script-src 'self' example.cloudfront.net;
             style-src 'self' 'unsafe-inline';
             img-src 'self' data: https:;
             connect-src 'self';
             frame-src 'self';
             form-action 'self';
             base-uri 'self';" />
    </customHeaders>
  </httpProtocol>
</system.webServer>
```

Secondary Fix

Deploy CSP in report-only mode first to identify violations without breaking functionality, then enforce after tuning.

```
<!-- Initial deployment: report-only mode at first -->
<add name="Content-Security-Policy-Report-Only"
   value="default-src 'self';
       script-src 'self' example.cloudfront.net;
```

```
        style-src 'self';
        report-uri https://acmetest.report-uri.com/r/d/csp/reportOnly;" />
// Startup.cs alternative with context.Response.Headers.Add("Content-Security-Policy",
```

Eliminate 'unsafe-inline' directives by refactoring inline scripts and styles to external files or using nonce-based CSP.

Acceptance Criteria

- Content-Security-Policy header present in all HTTP responses
- Policy restricts script-src, style-src, and default-src to trusted origins
- No application functionality broken by CSP restrictions
- CSP violations logged via report-uri for ongoing monitoring
- Policy tested against CSP evaluator tools (e.g., report-uri.com/home/analyse)

Details

Inspect HTTP response headers from any application page. Observe that no Content-Security-Policy header is present in the response.

```
# Response
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
# No Content-Security-Policy header present
```

HTTP response headers showing the absence of CSP

Without this header, the browser applies no restrictions on script execution sources, leaving XSS mitigation entirely dependent on application-level controls.

6. Information Disclosure in Detailed Error Messages

Vulnerability	Unhandled exceptions expose verbose error details, including file paths, developer names, and version information to unauthenticated users
Exploited via	Access /Account/TwoFactorAuth while unauthenticated to trigger unhandled exception and observe stack trace with internal paths and info
Immediate Risk	Internal system details disclosed to attackers, reducing reconnaissance effort for targeted exploits
Severity	**Informational**. No authentication required, affects error handling globally
Attack Surface	Any endpoint with unhandled exceptions, confirmed on TwoFactorAuthenticationSignIn
Fix	Configure custom error pages and disable verbose error output for remote users

Description

The application displays detailed exception information to remote users when unhandled errors occur. The TwoFactorAuthenticationSignIn page throws an uncaught exception when accessed by unauthenticated users, returning a full stack trace that includes the physical file path of the application (including developer username), source file names, line numbers, and framework version information. This indicates the application is configured with customErrors mode set to Off or RemoteOnly with inadequate error handling.

Impact

Verbose error messages accelerate attacker reconnaissance by revealing internal system architecture. Disclosed file paths expose directory structure and potentially developer naming conventions. Version information enables attackers to identify known vulnerabilities in specific framework versions without probing. Source file names and line numbers provide insight into the application structure that aids in crafting targeted exploits. While not directly exploitable, this information reduces the time and effort required to identify and exploit other vulnerabilities.

Context

Why this exists
Development-friendly error configuration was deployed to production. The customErrors mode is likely set to Off or RemoteOnly, and the specific code path lacks exception handling for the unauthenticated user scenario.

Architecture gap
Error handling strategy missing—no global exception handler sanitizes error output for production environments. Debug information intended for developers is exposed to all users.

Location

This vulnerability exists at:

- https://acmetest.example.com/Account/TwoFactorAuthenticationSignIn
- Configuration: web.config customErrors and httpErrors sections
- Potentially affects any endpoint with unhandled exceptions

Resources

- https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html
- https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html
- https://learn.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/h0hfz6fc(v=vs.100)

Remediation Approach

Primary Fix

Configure the application to suppress detailed errors for remote users and display generic error pages. Remove version headers from responses.

```
<!-- web.config -->
<configuration>
    <system.web>
        <!-- Disable version header -->
        <httpRuntime enableVersionHeader="false" />

        <!-- Show detailed errors only locally -->
        <customErrors mode="RemoteOnly" defaultRedirect="~/Error/General">
            <error statusCode="404" redirect="~/Error/NotFound" />
            <error statusCode="500" redirect="~/Error/ServerError" />
        </customErrors>
    </system.web>

    <system.webServer>
        <!-- Remove server header -->
        <security>
            <requestFiltering removeServerHeader="true" />
        </security>

        <!-- Configure IIS error handling -->
        <httpErrors errorMode="DetailedLocalOnly"
existingResponse="Replace">
            <remove statusCode="404" />
            <remove statusCode="500" />
            <error statusCode="404" path="/Error/NotFound"
responseMode="ExecuteURL" />
```

```
            <error statusCode="500" path="/Error/ServerError"
responseMode="ExecuteURL" />
        </httpErrors>
    </system.webServer>
</configuration>
```

Secondary Fix

Implement a global exception handler that logs detailed errors to ELMAH while returning sanitized responses to users.

```
// Global.asax.cs
protected void Application_Error(object sender, EventArgs e) {
    Exception exception = Server.GetLastError();
    // ELMAH automatically logs the exception with full details
    // Clear the error and redirect to generic error page
    Server.ClearError();
    Response.Redirect("~/Error/General"); }

// ErrorController.cs
public class ErrorController : Controller {
    public ActionResult General()     {
        Response.StatusCode = 500;
        return View(new ErrorViewModel          {
            Message = "An unexpected error occurred. Please try again later.",
            // Do not expose exception details
            ReferenceId Guid.NewGuid().ToString("N").Substring(0, 8)      });     }
}
```

Potentially add specific exception handling to the TwoFactorAuthenticationSignIn action to handle the unauthenticated user scenario gracefully.

```
[AllowAnonymous]
public ActionResult TwoFactorAuthenticationSignIn() {
    if (!User.Identity.IsAuthenticated)      {
        return RedirectToAction("Login", "Account");      }
        //etc
```

Acceptance Criteria

- Detailed stack traces are not visible to remote users on any endpoint
- Server version headers removed from HTTP responses (X-AspNet-Version, Server)
- Detailed errors still visible when accessing from localhost for debugging
- Error details logged to ELMAH for troubleshooting
- TwoFactorAuthenticationSignIn handles unauthenticated access gracefully

Details

Browse to /Account/TwoFactorAuthenticationSignIn while unauthenticated and not logged in. The web application displays an error message containing a detailed stack trace that discloses the folder path of the application, including the developer's name and source file name.

This information provides attackers with internal system details that can be leveraged to formulate more targeted attacks against the application.

Appendix - Web Application Security Assessment Methodology

Adversis brings extensive experience in assessing web applications. Adversis consultants hold CVEs across various web applications and common libraries and have experience building and breaking modern web application technologies such as React and GraphQL.

Adversis conducts a thorough assessment leveraging the OWASP Application Security Verification Standard (ASVS)[2] while identifying additional security issues, particularly business logic flaws. Our examination ensures comprehensive coverage of realistic vulnerabilities, meeting a minimum of OWASP ASV Level 1 standards and higher, as appropriate.

Adversis's approach to manual testing is exhaustive. We review and test action-performing (CRUD) requests within the application, thoroughly evaluating potential vulnerabilities with a focus on the OWASP Top 10.

Our primary tool, Burp Suite Professional, is integral to our testing process. It is selected for its capabilities in advanced vulnerability identification and manual testing features, which enable our team to pinpoint and address security weaknesses efficiently.

Adversis closely tracks OWASP Top Ten and believes that level one controls of the OWASP Application Security Verification Standard (ASVS)[3] are critical for any publicly facing application.

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server Side Request Forgery (SSRF)

In addition, the team considers your application's purpose, functionality, and data requirements, and assesses how data flows may be abused or functionality may be impacted.

[2] https://adversis.github.io/owasp-asvs/
[3] https://owasp.org/www-project-application-security-verification-standard/

Appendix - Risk Framework

Adversis leverages an industry-standard NIST (National Institute of Standards and Technology) threat matrix outlined in SP 800-30[4]. This matrix is a framework for analyzing risks and threats modified with input from the Factor Analysis in Information Risk (FAIR)[5] ontology. The matrix provides a structured and standardized approach to identifying, assessing, and prioritizing cybersecurity risks.

Likelihood of Threat Event	**Risk Rating**				
Very High	High	High	Very High	Very High	Very High
High	Medium	High	High	Very High	Very High
Significant	Medium	Medium	High	High	Very High
Moderate	Low	Medium	Medium	High	High
Low	Low	Low	Medium	Medium	Medium
Very Low	Low	Low	Medium	Medium	Medium
Loss Magnitude	**Neglible**	**Minor**	**Moderate**	**Major**	**Severe**

Event Frequency	**Probability Range**	**Calibration Anchor**
Very High	>76%	Likely exploitation within 12 months
High	51-75%	More likely than not to see exploitation at some point
Moderate	26-50%	It could go either way
Low	5-25%	Unlikely but possible
Very Low	<5%	All the stars would have to align

Loss Magnitude	**Loss Factors**	**Business Context**
Severe	>$1 Million	Financial impact, regulatory action, executive changes
Major	$100k - $1 million	Significant operational disruption, compliance violations
Moderate	$10k - $100k	Noticeable service degradation, limited data exposure
Minor	$1k - $10k	Minimal disruption, no sensitive data
Neglible	$0 - $1k	Best practice

Recommended Response Times

Critical	Remediation within 7 days
High	Remediation within 60 days
Medium	Remediation within 180 days
Low	Accept or remediate within 365 days

[4] https://csrc.nist.gov/pubs/sp/800/30/r1/final
[5] https://www.fairinstitute.org/what-is-fair