VidraSec

Penetration Test of the Example Web Application

Result Report

CLASSIFICATION: RESTRICTED

Organization:	Sample LLC
Recipients:	Alice Argyle (alice@example.com) Bob Bright (bob@example.com)
Date:	May 26, 2026
Version:	1.0
Project ID:	2025-06-20
Author:	

Table of Contents

Executive Summary

This report describes the results of the penetration test of the Example web application. The test spanned 5 person-days and was conducted using a *time-box* approach, aiming to identify as many vulnerabilities as possible within the available time. Higher-risk vulnerabilities were prioritized and tested first whenever possible.

The penetration test uncovered a significant privacy breach: any logged-in user can read the private profile data of every other account on the platform, including email addresses and personal settings. This requires no special knowledge or tools, just the ability to log in. The exposure covers the entire user base and should be urgently fixed.

Beyond this, a stored cross-site scripting vulnerability allows a malicious user to execute arbitrary code in the browsers of other users, including admins. This creates a realistic path to account takeover and could be combined with other findings to cause wider damage. The password policy also warrants attention: the current rules encourage predictable patterns rather than genuinely strong passwords, making accounts more susceptible to credential-based attacks.

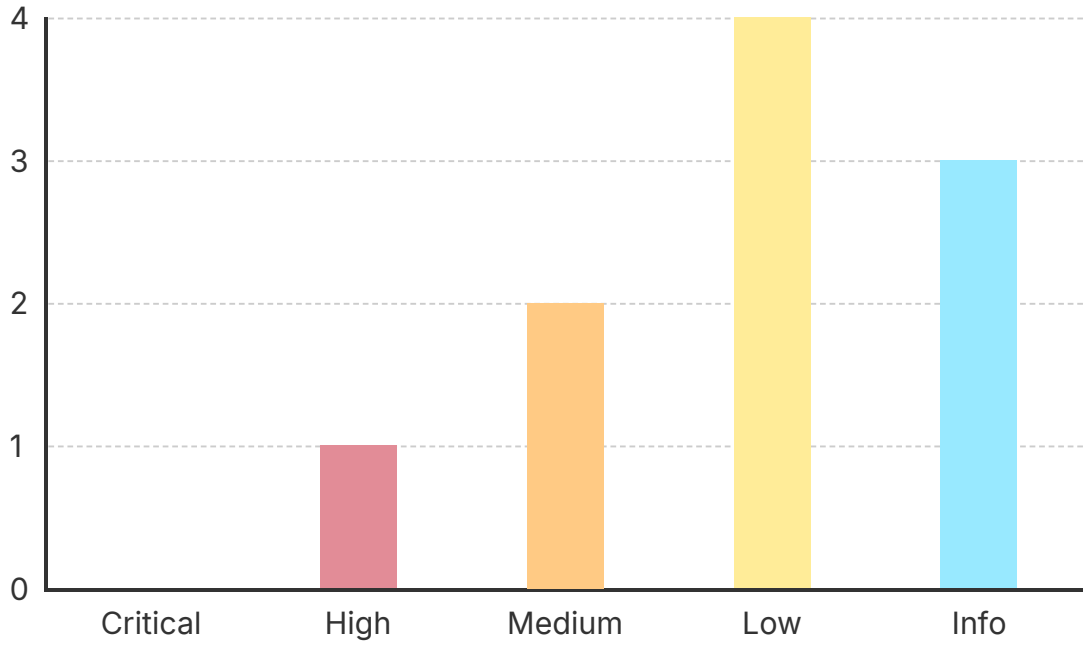
The remaining findings are of lower severity and in most cases require a chain of conditions to cause meaningful harm.

The vulnerabilities should be remediated. Afterwards the remediation should be verified and further penetration tests should be done in a regular interval to improve the security level of the application in a sustainable way.

Vulnerability Overview

Vulnerability	Severity
1. Broken Access Control	High
2. Cross Site Scripting (XSS)	Medium
3. Cross-Site Request Forgery (CSRF)	Medium
4. No Brute Force Protection	Low
5. Password Policy Should Be Improved	Low
6. User Enumeration Through Timing Attack	Low
7. Open Redirect	Low
8. Content Security Policy Is Missing	Info
9. Referrer-Policy Header Missing	Info
10. X-Frame-Options Header Missing	Info

Vulnerability Distribution by Severity



Test Coverage

A was conducted at Sample LLC with an extent of **5 person-days**.

To conduct the test, the penetration tester was provided with the following resources:

- 2 accounts with standard user privileges
- 2 accounts with admin privileges

The following systems were the targets of the penetration test:

System	Description
example.com	Example web application

The test was conducted 2026-02-01 – 2026-02-05.

Methodology

This penetration test was conducted using a *time-box* approach. This means that as many vulnerabilities as possible were uncovered within the available time. The process was prioritized, focusing first on vulnerabilities that are typically more problematic.

Severity Assessment

For each vulnerability, a technical severity level was determined. This does not always reflect the actual business risk posed by a vulnerability. For an accurate risk assessment, additional parameters such as the importance of the affected service should be considered alongside the technical severity level. Based on the risk, appropriate risk management measures can then be derived.

The following table provides an overview of the possible severity levels:

Severity	Meaning
Critical	The vulnerability is very problematic and should be addressed immediately. For example, it could allow for the complete takeover of a critical system.
High	The vulnerability is problematic and should be addressed as soon as possible. For example, it could allow for the takeover of a system.
Medium	This vulnerability could potentially be problematic and should be analyzed. For example, it could allow sensitive information to be read.
Low	This vulnerability should be analyzed. For example, it could be exploited in combination with other vulnerabilities or disclose internal information.
Info	This type of vulnerability is not problematic by itself. Examples would include measures that could further enhance security.

This report uses the Common Vulnerability Scoring System version 4.0 to assess the severity of vulnerabilities. CVSS is owned by FIRST.Org, Inc. (FIRST), a non-profit organization based in the USA.

This document describes CVSS in detail: https://www.first.org/cvss/v4-0/cvss-v40-specification.pdf

Vulnerabilities

1. Broken Access Control

High (7.1)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

Description

“Broken Access Control” is a vulnerability category where users can access functionality in the application that they should not be allowed to access. In many cases, this is simply due to implementation oversights in authorization checks. The underlying root cause is often insufficient documentation of who should have access to which functionality.

“Broken Access Control” ranks #1 in the OWASP Top Ten [1], meaning it is the most common vulnerability category in web applications. One reason is that there is limited technical automation for remediation. The authorization model must be designed by someone with deep knowledge of the application and use cases, and then implemented without mistakes.

Insecure Direct Object Reference (IDOR)[2] and Broken Object Level Authorization (BOLA)[3] are specific forms of Broken Access Control [4]. In these cases, files or objects can be accessed by guessing a random or sequential object ID.

Concrete Vulnerability

An authenticated user can access other users’ profile data by manipulating the user ID in the API path. Sending a `GET` request to `/api/users/{id}/pro`

`file` returns the full profile of the specified user — including email address, display name, and account settings — regardless of whether the authenticated user owns that account. By iterating through sequential numeric IDs, an attacker can enumerate the profiles of all registered users.

Countermeasures

As a first step, the specific identified access-control issues should be fixed.

As a second step, documentation should be created that defines which users or roles have which access rights to which functions. These designed permissions must then be implemented. As a final step, permissions should be reviewed by an independent person (e.g. via penetration test).

As an advanced measure, it is recommended to additionally document permissions in a machine-readable format. This makes it possible to build automated test cases to detect permission problems as early as possible.

As an additional safeguard, IDs can be chosen randomly (UUID v4[5]). Access control still needs to be implemented, but exploitation becomes harder because a malicious actor must first discover the random object ID.

---

1. OWASP. OWASP Top Ten: https://owasp.org/www-project-top-ten/
2. OWASP. Insecure Direct Object Reference Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html
3. OWASP. API1:2023 Broken Object Level Authorization: https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/
4. Mustafa Ergin. What is the difference: BOLA vs IDOR: https://shoutrange.com/insights/bola-vs-idor-what-is-the-difference
5. Wikipedia. Universally unique identifier: https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_(random)

2. Cross Site Scripting (XSS) Medium (6.3)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:P/VC:N/VI:N/VA:N/SC:H/SI:H/SA:N

Description

General Description

Cross-Site Scripting (XSS) is a category of vulnerabilities that allows malicious users to inject executable JavaScript code into a web application. The injected JavaScript code can be used to manipulate the web application arbitrarily, perform actions in the victim's context, or read the victim's secrets. A known tool for taking over the victim's browser via malicious code is beef[1]. In the worst case, the vulnerable website could therefore be abused by an attacker to target users of the site.

Reflected XSS

In a *Reflected XSS* vulnerability, a request to the website can be manipulated so that executable JavaScript code is returned in the response. One example is a search function that echoes the original search query. In the worst case, an attacker can generate a link that sends such a request. When the victim opens the link, the malicious code is executed. The victim only has to be tricked into clicking the malicious link. This is especially easy in this case because the link points to the real, trusted website.

Stored XSS

With *Stored XSS*, JavaScript is also injected into the website. In this case, the attacker can persistently inject JavaScript code, meaning it is also stored, for example, in a database. A guestbook is a typical example. An attacker could

submit a guestbook entry containing JavaScript. When users visit the guestbook, that JavaScript code is executed.

Concrete Vulnerability

A stored XSS vulnerability was identified in the “Display Name” field of the user profile settings (`POST /api/profile`). When a user sets their display name to a payload such as `<script>alert(document.cookie)</script>`, this script is stored without sanitization and subsequently executed in the browser of every user who visits a page displaying that name — for example, the admin panel’s user management view or any comment thread authored by the affected user.

Countermeasures

The “Display Name” input must be sanitized before being stored and HTML-encoded before being rendered in any page context. The React frontend should rely exclusively on text-node rendering (e.g., `textContent`) rather than `innerHTML` for user-supplied values.

In general, modern web frameworks provide good protections against XSS and should be used. In most cases, output encoding should be preferred over input encoding as a mitigation. This keeps input unchanged while preventing code execution when output is rendered.

More details on XSS countermeasures can be found in [2].

---

1. beefproject. beef: https://github.com/beefproject/beef
2. OWASP. Cross Site Scripting Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

3. Cross-Site Request Forgery (CSRF) Medium (5.1)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:N/VI:N/VA:N/SC:N/SI:N/SA:L

Description

Cross-Site Request Forgery (CSRF) is a type of vulnerability in which a malicious website visited by a victim can send background requests to the vulnerable website on behalf of that victim.

An example would be a victim who is an admin in a web application. The victim is logged into this web application, and an active session cookie is stored in the browser. The victim now visits a malicious website. This malicious website can then use JavaScript in the background to send HTTP requests to the vulnerable web application (e.g., creating a new account in the vulnerable web application). Depending on the configuration and the type of HTTP request, the victim's browser automatically includes the victim's session cookie, so the HTTP requests are executed in the victim's context.

The tested application is already relatively well protected against CSRF, because requests requiring an `application/json` content type are not susceptible to CSRF attacks.

In addition, HTTP request methods are used correctly in the application. For example, GET requests do not trigger state changes. This is important because this method is especially susceptible to CSRF.

The `/api/auth/logout` endpoint accepts GET requests and is not protected by a CSRF token. Because GET requests are excluded from the additional protections provided by `SameSite=Lax`, a cross-site attacker can force a

victim to log out by embedding an invisible resource request such as `<img src="https://example.com/api/auth/logout">` on any page. When a logged-in user visits the attacker-controlled page, the session is silently terminated.

Countermeasures

The classic countermeasure against CSRF is to set CSRF tokens. The application generates a random token when the page is loaded. HTTP requests must include this token. For a successful CSRF attack, the token would have to be guessed.

In addition, it must be ensured that the correct HTTP methods are used. A GET request must never trigger a state change in the application.

Further information about the attack and recommended countermeasures can be found in [1]. Most web frameworks provide support for implementing CSRF countermeasures.

---

1. Mozilla. Cross-site request forgery: https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/CSRF

4. No Brute Force Protection Low (2.3)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:H/AT:N/PR:N/UI:P/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

Description

The application currently lacks protection against brute-force attacks.

This was tested by attempting to log into a single account with an incorrect password 100 times in rapid succession. All requests were processed without restriction. Immediately following these attempts, a login with the correct credentials was successful, confirming that the account was neither locked nor rate-limited.

Countermeasures

The most straightforward defense against brute-force attacks is account lockout temporarily disabling an account after a set number of failed attempts. While effective, this can be abused to trigger Denial of Service (DoS) attacks, where an attacker intentionally locks out legitimate users.

Alternatively, rate limiting or request throttling can be used to artificially increase response times after multiple failed attempts. This slows down automated attacks to a negligible speed while still allowing valid users to authenticate (with a short delay).

A more sophisticated approach involves device cookies. These are long-lived tokens generated upon a successful login that link a specific device to a user account. Logins originating from a “known” device with a valid cookie can bypass certain brute-force triggers, reducing friction for legitimate users. Detailed implementation guidance is available in [1].

> More information on how to block brute force attacks can be found in [2].
>
> Ultimately, there is no universal standard for brute-force protection. Security teams must implement countermeasures tailored to the specific attack surface and the risk profile of the application.

---

1. OWASP. Slow Down Online Guessing Attacks with Device Cookies: https://owasp.org/www-community/Slow_Down_Online_Guessing_Attacks_with_Device_Cookies
2. OWASP. Blocking Brute Force Attacks: https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks

5. Password Policy Should Be Improved Low (2.3)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:H/AT:P/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

Description

A password policy should help users set a secure password that cannot be easily guessed by attackers.

The following password policy is currently set:

- Minimum length: 8 characters
- Must contain at least one uppercase letter, one lowercase letter, one digit, and one special character

The enforced complexity rules lead to predictable password patterns (e.g., `Password123!`) that satisfy the policy but are well-known to attackers. At the same time, no blocklist exists to reject commonly used passwords.

The following password could be set: `Password123!`.

Countermeasures

The National Institute of Standards and Technology (NIST) has published widely accepted guidance for password policies (see [1]).

The key points are:

- At least 8 characters (this can be higher depending on protection requirements)

- Use modern hash algorithms specifically designed for password storage (e.g., Argon2)[2]
- Presence of protection against brute-force attacks
- Blocklist to prevent users from setting known weak passwords (e.g., `Password123!`) (for example, the solution in [3] can be used)
- **No** password complexity requirements
- **No** forced regular password changes

Important: For a new password policy to actually take effect, all users must be forced to change their password after the policy is updated. This is because the password policy is only applied when a password is set again. Alternatively, during login it could be checked whether the current password complies with the policy and the user could be prompted to change it.

---

1. NIST. Digital Identity Guidelines: https://pages.nist.gov/800-63-3/sp800-63b.html
2. Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications: https://www.cryptolux.org/images/0/0d/Argon2.pdf
3. Troy Hunt. Pwned Passwords: https://haveibeenpwned.com/Passwords

6. User Enumeration Through Timing Attack

Low (2.3)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:H/AT:N/PR:N/UI:P/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

Description

Even though the application returns the same generic response if an existent or non-existent user is supplied, it is possible to determine whether an email address is valid or not by comparing the HTTP response times.

Password Reset

This vulnerability was confirmed on the password reset form (`POST /api/auth/reset-password`).

When a non-existent user was supplied to the password reset function the response time was approximately 45 ms.

When an existent user was supplied, the response time was always more than 430 ms, due to a synchronous call to the external email delivery service.

Login

This vulnerability was confirmed on the login form (`POST /api/auth/login`).

When a non-existent user was supplied, the response time was approximately 18 ms.

When an existent user was supplied, the response was approximately 210 ms, due to bcrypt password hashing being performed only for existing accounts.

Both occurrences of this vulnerability should be retested from a system closer to the target (e.g., on the local network), to ensure that no hops in-between caused the increase in response time.

Countermeasures

All requests should have similar response times, regardless of whether the supplied username exists or not.

Login

Most of the time, the culprit of different response times is that a check is first done to determine whether a user exists, and only afterwards is password hashing performed. To counter this, every supplied password should be hashed, because hashing is, most of the time, the most time-consuming operation. Only afterwards should there be a check to determine if the user exists. The application should then still return a generic error message, not revealing whether the check failed because of the password or username.

Password Reset

Often, password reset forms take longer to respond to password reset requests with a valid email address, because a third-party service is called, and the application waits for the response. This should either be changed so that the response is sent immediately and the third-party service is called asynchronously, or both responses should be delayed artificially to make timing attacks impossible.

7. Open Redirect

Low (2.1)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:H/AT:N/PR:N/UI:A/VC:N/VI:N/VA:N/SC:L/SI:N/SA:N

Description

An open redirect is a vulnerability where the application triggers a redirect whose target can be manipulated. This allows a malicious person to abuse the application to lure a victim to a potentially malicious website. This misuses and may damage trust in the affected web application.

Concrete Vulnerability

The login endpoint (`GET /login`) accepts a `next` URL parameter to redirect the user to a specified page after successful authentication. This parameter is not validated against an allowlist and accepts arbitrary external URLs. An attacker can craft a link such as `https://example.com/login?next=https://malicious.example` and distribute it as a seemingly legitimate login link. After authenticating on the genuine login page, the victim is silently redirected to an attacker-controlled website, where credentials or session tokens could be harvested.

Countermeasures

It should not be possible to specify an arbitrary external URL as a redirect target through these endpoints. If that is sufficient, only relative (internal) URLs should be allowed as redirect targets. If that is not sufficient, an allowlist of external URLs or domains should be defined that are allowed as redirect targets.

> More information about this vulnerability and possible countermeasures can be found in [1].

1. OWASP. Unvalidated Redirects and Forwards Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html

8. Content Security Policy Is Missing Info (0.0)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:H/AT:P/PR:N/UI:P/VC:N/VI:N/VA:N/SC:N/SI:N/SA:N

Description

Content Security Policy (CSP) is an advanced security measure that uses an HTTP header to define which sources a website is allowed to load external resources from or execute code from.

This primarily makes the exploitation of cross-site scripting vulnerabilities harder, because by default only explicitly allowed JavaScript code may run. Inline script code or code from unknown sources is no longer executed. Even if a malicious person manages to inject harmful JavaScript code into the website, it will not be executed.

In addition, it can also be used to control whether the website may be embedded in an `iframe` and from which sources resources may be loaded.

Currently, no *Content Security Policy* is set on the affected system.

Countermeasures

Implementing a *Content Security Policy* (CSP) is not trivial because it must be tailored individually for each web application. Depending on the application architecture, code changes may also be required to enable CSP usage.

In general, a *Content Security Policy* should not include `unsafe` attributes, as these disable important security features.

> Details on implementing a CSP can be found in [1], [2], and [3].

---

1. Lukas Weichselbaum. Mitigate cross-site scripting (XSS) with a strict Content Security Policy (CSP): https://web.dev/articles/strict-csp
2. Mozilla. Content Security Policy (CSP): https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP
3. OWASP. Content Security Policy Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html

9. Referrer-Policy Header Missing Info (0.0)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:N/SI:N/SA:N

Description

The *Referrer-Policy* HTTP header defines whether the browser sends the `Referer` (the page where a link was clicked) to the target when opening a link. By default, this is the case.

This means that if a user clicks a link to `https://target.example` on `https://source.example`, the `Referer https://source.example` is sent in the HTTP request to `https://target.example`. As a result, `https://target.example` learns that the user was previously on `https://source.example`, which can affect user privacy.

The `Referrer-Policy` header was not set in HTTP responses from `example.com`.

Countermeasures

Following the principle of data minimization, the *Referrer-Policy* header should be set to `no-referrer`, `origin`, or `same-origin`. More information about this header can be found in [1].

1. Mozilla. Referrer-Policy: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Referrer-Policy

10. X-Frame-Options Header Missing Info (0.0)

Affected Systems

- example.com

CVSS Vector

CVSS:4.0/AV:N/AC:H/AT:N/PR:N/UI:A/VC:N/VI:N/VA:N/SC:N/SI:N/SA:N

Description

The *X-Frame-Options* header prevents *clickjacking* attacks[1]. In such attacks, a malicious website embeds the vulnerable website in an *iframe*, which can even be invisible. The victim is lured to the malicious site and tricked into clicking specific elements. In reality, however, the embedded vulnerable page receives these clicks, so the victim unknowingly performs actions there.

This vulnerability is particularly problematic when the affected page has an authenticated area where actions can be performed. The tested application does have such an authenticated area. However, no high-impact actions were identified that could be meaningfully triggered through clickjacking in the current application, so the risk is rated as informational. Setting this header is still recommended as a defense-in-depth measure.

Countermeasures

The *X-Frame-Options* header should be set to `DENY` or `SAMEORIGIN`[2]. In addition, the `frame-ancestors` directive can be defined in the *Content Security Policy*, which provides similar protection.[3]

---

1. Mozilla. Clickjacking: https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/Clickjacking

2. Mozilla. X-Frame-Options: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/X-Frame-Options

3. Mozilla. CSP: frame-ancestors: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Security-Policy/frame-ancestors

Imprint

VidraSec e.U. – https://www.vidrasec.com/
Address: Almesbergerweg 3, 4160 Aigen-Schlägl, AUSTRIA
Commercial Register Number: 623204b
Commercial Court: Regional Court Linz
Trade Supervisory Authority: BH Rohrbach
VAT ID: ATU80334229

Contact details
Email: martin@vidrasec.com
Phone: +43 670 3081275

Bank details
Bank: REVOLUT BANK UAB
IBAN: LT76 3250 0126 5399 4118
BIC: REVOLT21