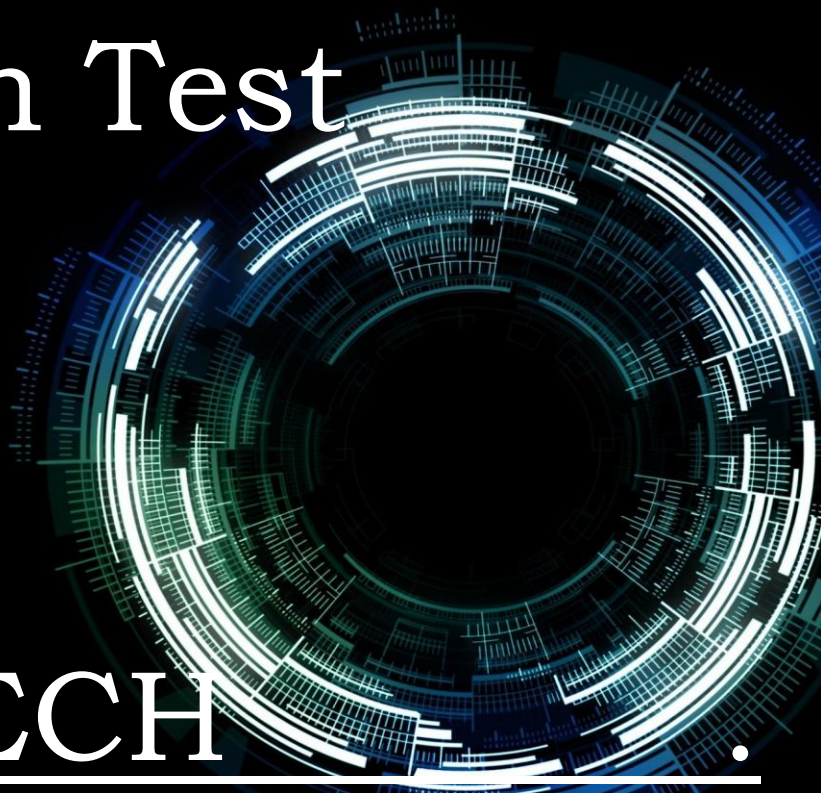
ECOM SCHOOL

Urban Online Academy & Network

Penetration Test Report



HACKER TECH .

September, 2025

Black Box Penetration Testing

This legal disclaimer regulates the usage of this Penetration Testing report and the content of this report. The contents of this document have been prepared from the information collected by the "ECOM" research team. It is worth noting that the information offered in this report constitutes a general framework for security screening and is not intended to ensure protection or compliance with any requirements and guidelines of any authority and the like. Nor can it be concluded that compliance with the recommendations of this report will ensure that the subject of the test is protected and that it does not contain any additional vulnerabilities. Under no circumstances will the Penetration Testing team and the " ECOM " team be responsible for any damage directly or indirectly caused by this test of any kind. The contents of this entire document are confidential and completely confidential and are the intellectual property of the " ECOM " Research Team and may be used only internally for educational purposes only. In addition, it is worth noting that the structure of this document is confidential and is the intellectual property of the " ECOM " research team and may not be used if you are not a member of this team.

.

TABLE OF CONTENT

REPORT STRUCTURE

This report contains three different sections:

1. **Executive Summary** - This section includes a brief description of the content of the work as well as a list of the main findings that constitute potential for damage and, as a result, require the organization to take corrective steps in our view.

2. **Details of the tests** - This section details all the tests performed by division into the various areas as well as a description of the information collected in the survey. This section also lists all the findings of the exam, the description of the risks as a result of the findings, and the recommendations for implementation based on the accumulated experience of ECOM.

3. **Appendices** - Brief of the methods used during the penetration test with additional explanation about our rating system fix effort.

ABOUT THE EDITOR

Arad Ceizler, is a penetration testing student.

Arad Ceizler is a penetration testing student with a strong background in IT support and computer networking. Through his role as a Helpdesk specialist, he has gained practical experience in troubleshooting, system maintenance, and communication within technical environments. He is currently studying at Ecom College in a program that covers both infrastructure and application penetration testing, developing practical skills in ethical hacking, vulnerability assessment, and security analysis.

www.linkedin.com/in/aradceizler

EXECUTIVE SUMMARY

BACKGROUND

The "ECOM" Cyber Security Team was asked to perform an applicative penetration test for the HACKER TECH on September 2025.

The test scenarios performed included attempts to infiltrate the customer's services, taking the advantage of the built-in weaknesses, taking into account the type of applications/operating systems and the type of components with which the customer works.

The test was performed to detect vulnerabilities that could put HACKER TECH at risk and to simulate a situation where an attack occurs while making maximum use of the resources available to the attacker.

This report includes a description of all the vulnerabilities found, a general explanation of them, Proof Of Concept and other findings for the customer to be able to harden his services and increase his level of security.

This test was performed from Arad's home office Bethlehem Street 27, Haifa, Israel, by the Penetration Testing team of "ECOM".

This test was performed using a Black box Penetration Test methodology, and the test content was determined as part of the delineation, both in terms of the topics and components to be tested and the scope of resources that will be allocated to the test. Thus, the test may not detect all the infrastructural and applicative exposures of the client network.

The findings set forth in this document are correct as of the date of the test. Any applicative or infrastructural change made after the end of the test may affect the security level of the client.

It is worth noting that the official contact person on behalf of the company is HACKER TECH and all the tests were matched with him.

PROJECT DESCRIPTION

SCOPE & TARGETS

In advance with the client, the test team was given the following goals:

No.	Target Address	Extra Details
1	Techie-world.xyz	

This test contains a number of infrastructural / applicative test methodologies in order to examine the level of risk of the information that is output in the identified systems.

As part of this examination, the following were examined (**Delete the unnecessary**):

<In case of applicative examination>

- A number of code injection techniques at both the client and server level that can significantly compromise the information stored in this system.
- OWASP TOP 10 includes a variety of vulnerabilities and advanced attack techniques.
- Check for system bugs that can lead to malicious actions at the user level.

TEST LIMITATIONS

Due to time limitations, a comprehensive assessment of infrastructure-level vulnerabilities was not performed; therefore, there may be infrastructure issues that could expose further information about the server.
Additionally, stored and DOM-based Cross-Site Scripting (XSS) checks were not performed after the discovery of reflected XSS vulnerabilities; these vectors remain unvalidated and may require follow-up testing.

SUMMARY & ASSESMENT

During the test it was found that an attacker can perform RCE (Remote Code Execution) using a chain of attacks: brute-forcing a standard user credential, manipulating session cookie values to escalate to an administrative account, and accessing a sensitive mpdf file whose embedded annotation exposed an authentication hash that yielded a command shell, along with other misconfiguration defects.

The penetration testing lasted 14 days, and resulted several vulnerabilities such as Reflected XSS, HTML Injection and CSRF. An attacker that is exploiting these bugs may gain Remote Rode Execution leading to root access, enabling command execution and access to sensitive data. This may damage the organization's reputation and may put the organization and its clients at risk.

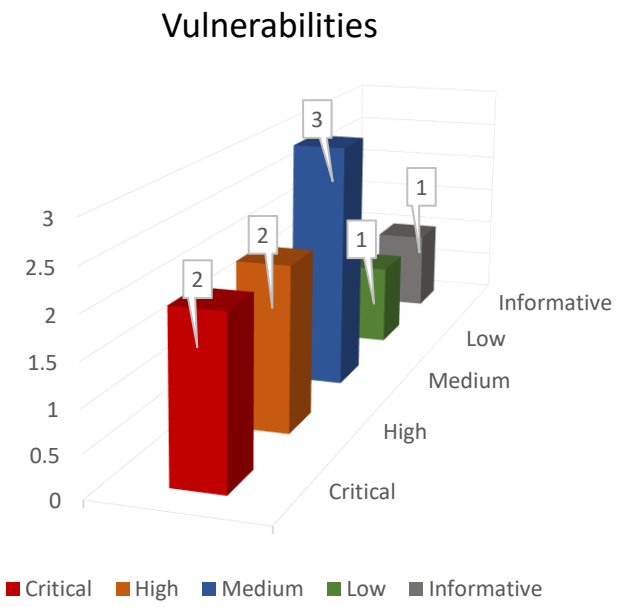
CONCLUSIONS

From our professional view, the security level exists in the client's systems is now on High.

The was rated as mentioned before due to the existence of multiple vulnerabilities such as RCE and Path Traversal leading to root access and exposure of sensitive information, which may put the organization and its clients at risk.

Exploiting most of the vulnerabilities mentioned above requires a High technical knowledge.

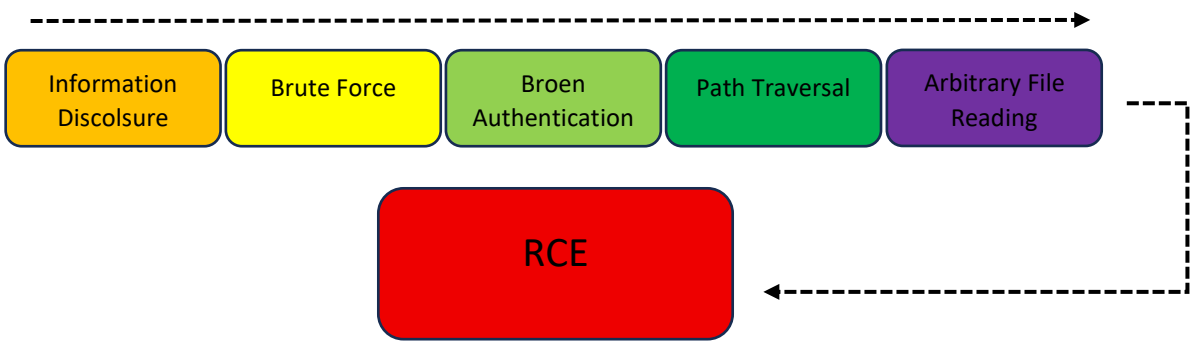
<note – change the graph according to the findings>



ATTACK TREE FOR COMPLEX SCENARIOS

The following diagram describes each complex attack scenarios that can be applied in the client's system.

ATTACK SCENARIO NO.001 - Remote Code Execution Kill Chain



SETTING GOALS AND OBJECTIVES

The following objectives were defined for intrusion testing operations as objectives of paramount importance.

In the case of PT work:

- Search for ***low hanging fruits*** – (**ACHIEVED**)
- Finding a ***number of vulnerabilities*** that could endanger the target – (**ACHIEVED**)

In case of applicable external work:

- Performs a vulnerability combination ***perform a complex attack*** to maximize the attacker's abilities - (**ACHIEVED**)
- Exposing the target to the ability to ***run code remotely*** - (**ACHIEVED**)

In the case of infrastructure internal work:

- Obtaining ***Domain Admin privileges*** in the target's environment - (**NOT ACHIEVED**)
- Performing ***Lateral Movement***, exposing and exploiting additional positions - (**NOT ACHIEVED**)
- ***Utilization of protocols*** in favor of the attacker - (**NOT ACHIEVED**)

IDENTIFIED VULNERABILITIES

VULN-001 Information Disclosure (INFORMATIVE)

The application's front-end source discloses the password format: passwords are numeric and 4–6 characters long. This client-side disclosure significantly reduces password entropy and directly lowers the effort required for brute-force attacks.

VULN-002 Brute Force (MEDIUM)

During the engagement the tester intercepted the application's login POST request using Burp Suite and submitted that request to Burp Intruder to perform an automated credential-guessing (brute-force) attack against the password parameter. Due to a weak client-exposed password policy (numeric, 4–6 characters) and absent account lockout/rate-limiting, the automated attempts succeeded in recovering a valid standard-user credential.

VULN-003 Broken Authentication (CRITICAL)

During the engagement the tester discovered a Broken Authentication issue in session handling: a client-modifiable session cookie (without server-side integrity checks) was altered to impersonate an administrative account, allowing login as **admin** without valid admin credentials.

VULN-004 Arbitrary File Reading (HIGH)

The application accepts a user-supplied filename for PDF annotations (the annotation file parameter) and loads it without proper validation or path sanitization. The supplied screenshot shows mpdf.php being loaded via this parameter, indicating the ability to read local files on the server.

VULN-005 Path Traversal (HIGH)

The application accepts a user-supplied filename for PDF annotations and processes it without sufficient server-side validation or path sanitization, allowing path traversal that can be used to access files outside the web root.

VULN-006 Remote Code Execution (CRITICAL)

An exploit chain allowed the tester to obtain a remote command shell. A discovered annotation referenced mpdf.php which contained an authentication hash; submitting that value resulted in shell access, from which the tester executed commands and escalated privileges to root, exposing sensitive data.

VULN-007 Cross Site Scripting (MEDIUM)

A Reflected Cross-Site Scripting (XSS) vulnerability was identified in an input field of the site's contact form. User-supplied input is reflected back to the page without proper output encoding/escaping, allowing arbitrary JavaScript execution in the browser of visiting users.

VULN-008 Cross-Site Request Forgery (MEDIUM)

The contact-form endpoint accepts and processes POST requests without anti-CSRF protection. There is no server-validated CSRF token or reliable origin check in place, allowing cross-site requests to be forged and executed in the context of authenticated users.

VULN-009 HTML Injection (LOW)

The application reflects user-supplied content into the page as raw HTML without proper output encoding or sanitization, resulting in HTML Injection. Attackers may inject arbitrary HTML markup that modifies page structure or content.

FINDING DETAILS

VULN-001 Information Disclosure

CVSS

CVSS:5.3/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | Medium	Probability | High	Severity | Medium	Fix Effort | Low

DESCRIPTION

Information disclosure is a class of security weakness where an application unintentionally reveals internal, sensitive, or otherwise useful information to parties that should not have access to it. Such disclosures can appear in many forms — for example in client-side source (HTML/JS), server responses (error messages, stack traces), configuration files, metadata, HTTP headers, logs, directory listings or publicly readable backups — and they increase an attacker's knowledge about the target environment. The practical effect ranges from merely exposing implementation details to materially lowering the effort required for further attacks (e.g., enabling credential guessing, targeted exploitation of known component vulnerabilities, path traversal, or social-engineering).

Information disclosure can be categorized by how and where the data is revealed, for example:

- **Client-side exposures** — Source HTML, JavaScript comments, inline configuration, or UI hints (e.g., password policy, debug comments, hard-coded endpoints/tokens) that are visible via view-source or DevTools.
- **Server-side response leakage** — Verbose error pages, stack traces, SQL errors or debug output returned to clients.
- **Configuration and file artifacts** — Exposed .env/backup/config files, directory listings, or publicly accessible files that disclose credentials, keys, or internal paths.
- **Header and metadata leakage** — Overly informative HTTP headers, permissive CORS settings, or other protocol-level metadata revealing software versions or internal services.
- **Logging and telemetry leakage** — Unprotected log files, monitoring consoles, or diagnostic endpoints that expose sensitive runtime information.

The severity of an information disclosure depends on the sensitivity and scope of the data exposed and on whether compensating controls exist; even apparently minor details (password format, directory structure, or component versions) can be high-value if they enable or accelerate subsequent exploitation.

PROOF OF CONCEPT

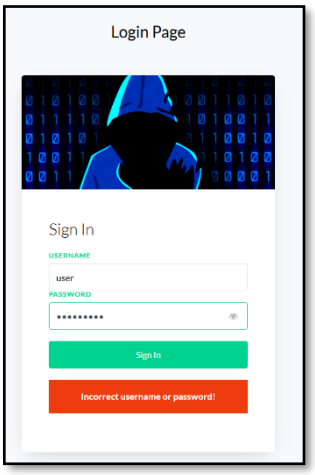


*Figure 1 - Login page screenshot showing the **"Incorrect username or password!"** message after submitting a valid username with an incorrect password, confirming informative error feedback that aids account reconnaissance.*

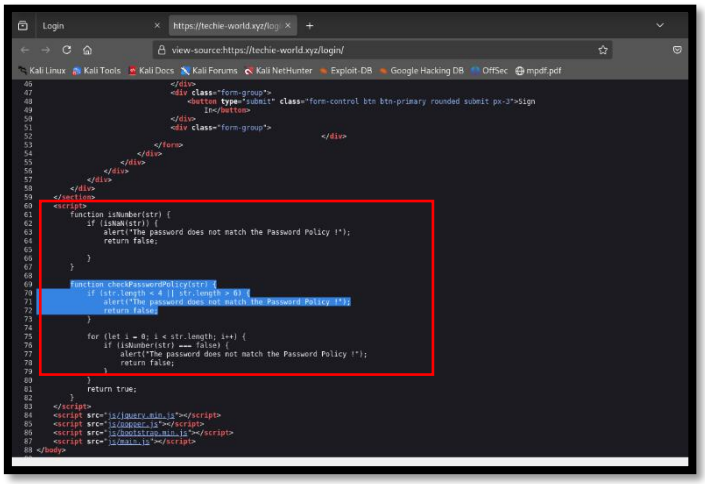


Figure 2 - View-source snippet showing the client-side checkPasswordPolicy function that reveals the password format/length constraints

DETAILS

During testing I performed simple authentication probes against the login form: submitting a non-existent username returned **"user not found"**, while submitting a known username (user) with a wrong password returned **"Incorrect username or password!"** (see Figure 1). To look for additional clues I opened the page source (Ctrl+U) and searched the HTML/JavaScript for authentication-related identifiers.

In the source I found a client-side password validation routine (checkPasswordPolicy) that restricts passwords to **digits only** and a length of **4–6 characters** (see Figure 2). Because this policy is visible in the delivered client code (and server-side enforcement was not observed in this read-only step), the disclosure significantly reduces password entropy and, together with the informative error messages, materially aids automated credential-guessing (brute-force).

Evidence (screenshots of the view-source and login responses) has been captured and stored in the protected technical appendix. Safe reproduction: open the login page, try a non-existent username and a known username with a wrong password to observe the responses (Figure 1), then press Ctrl+U and search for checkPasswordPolicy (Figure 2). Full exploit transcripts and sensitive artifacts are withheld from the main report and provided only in the protected appendix.

RECOMMENDED MITIGATIONS

1. Uniform Error Messages
   The application currently reveals whether a username exists through distinct error messages ("User not found" vs. "Incorrect password"), which enables username enumeration. To mitigate this, configure the authentication mechanism to always return a generic error message such as *"Invalid username or password"* for all failed login attempts. Ensure both the HTTP status code and the response body remain consistent, preventing attackers from distinguishing between invalid usernames and incorrect passwords.

   Reference: OWASP Authentication Cheat Sheet

2. **Rate Limiting and Account Lockout**
   No rate limiting or account lockout mechanisms were observed during testing, which allows brute-force attacks against the reduced password space. To mitigate this, implement per-user and per-IP rate limiting and introduce temporary account lockouts after multiple failed attempts (e.g., exponential backoff). Additional measures such as CAPTCHA after repeated failures and Multi-Factor Authentication (MFA) can further strengthen protection against brute-force attempts.

   Reference: OWASP Blocking Brute Force Attacks

VULN-002 – Brute Force

CVSS

CVSS:5.3/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | Medium	Probability | High	Severity | Medium	Fix Effort | Low

DESCRIPTION

Brute-force attacks are an authentication-targeted technique in which an attacker systematically attempts many credential combinations (passwords, PINs, or token values) to gain unauthorized access to user accounts or services. Brute-force may be performed against online authentication endpoints (online brute-force) or offline against stolen hashes/password dumps (offline brute-force). The practical feasibility of a brute-force attack depends on several factors: the size of the credential keyspace (password complexity and length), whether usernames can be enumerated, the presence (or absence) of rate limiting and account lockout controls, and any additional protective controls such as multi-factor authentication (MFA), CAPTCHA, or progressive throttling.

Brute-force risks present in multiple forms and contexts, for example:

- **Online credential guessing** — Automated submission of password lists or generated combinations to a login endpoint; effectiveness increases when the server exposes password format constraints (e.g., numeric only, fixed length).
- **Username enumeration + targeted guessing** — Distinct authentication responses (or timing differences) allow an attacker to identify valid accounts and then focus guessing efforts on those accounts, greatly improving success probability.
- **Credential stuffing** — Reuse of leaked username/password pairs from other services; small or predictable password formats make stuffing far more likely to succeed.
- **Offline brute-force** — Given stolen password hashes, attackers can try extremely large numbers of guesses offline (often accelerated with GPUs); weak hashing or lack of salts magnifies this risk.
- **Distributed / scale attacks** — Attackers may distribute requests across many IPs or botnets to evade per-IP rate limits; robust per-account controls are needed to mitigate this.

The practical effect of successful brute-force ranges from simple account takeover (access to non-sensitive user data) to full compromise of privileged accounts and downstream escalation (unauthorized transactions, data exfiltration, administrative changes). Key enabling factors that materially lower attacker effort include exposed password policies (client-side or server-returned constraints), informative error messages that permit username enumeration, and absent or ineffective throttling/lockout mechanisms.

PROOF OF CONCEPT

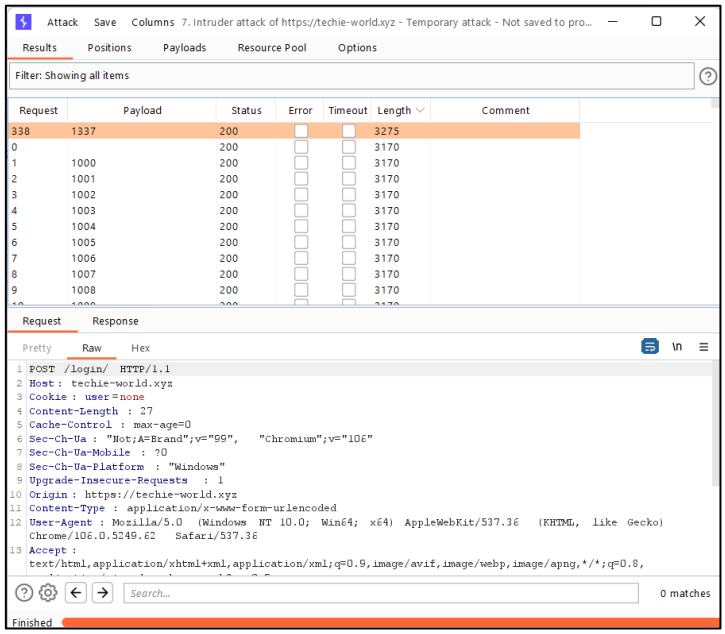


Figure 1 - Burp Intruder brute-force attempts against the /login endpoint (numeric payloads).

DETAILS

During testing a valid login request for the account user was captured via proxy and used as the Intruder template. Burp Suite Intruder was configured with the password form parameter as the payload position, and a numeric payload set covering all 4–6 digit combinations was loaded and executed against the endpoint.

All automated attempts returned consistent HTTP responses and no account lockout or throttling was observed during the run. The automated attack successfully discovered a valid credential for the tested account (user:1337), confirming that, under the current configuration, the endpoint is susceptible to practical online brute-force when combined with an exposed, small password keyspace.

Attack logs, the captured POST request and related artifacts were recorded and retained in the protected technical appendix; sensitive items (discovered credentials and full request/response logs) are withheld from the main report. Immediate remediation should focus on per-account/IP rate limiting and unified authentication failure responses to prevent targeted credential-guessing.

RECOMMENDED MITIGATIONS

1. **Uniform Error Messages**
   The application currently reveals whether a username exists through distinct error messages (“User not found” vs. “Incorrect password”), which enables username enumeration and improves brute-force efficiency. To mitigate this, configure the authentication mechanism to always return a generic error message such as *“Invalid username or password”* for all failed login attempts. Ensure both the HTTP status code and the response body remain consistent, preventing attackers from distinguishing between invalid usernames and incorrect passwords.

Reference: [OWASP Authentication Cheat Sheet]

2. **Rate Limiting and Account Lockout**
   No rate limiting or account lockout mechanisms were observed during testing, which allows brute-force attacks against the reduced password space. To mitigate this, implement per-user and per-IP rate limiting and introduce temporary account lockouts after multiple failed attempts (e.g., exponential backoff). Additional measures such as CAPTCHA after repeated failures and Multi-Factor Authentication (MFA) can further strengthen protection against brute-force attempts.

Reference: OWASP Blocking Brute Force Attacks

3. **Server-Side Password Policy and MFA**
   The password policy was exposed in client-side code, showing that passwords are restricted to 4–6 digits, which significantly reduces the keyspace. To mitigate this, enforce password complexity and length requirements strictly on the server side (e.g., 12+ characters, allow passphrases). In addition, require Multi-Factor Authentication (MFA) for administrative and high-risk accounts, which makes brute-forcing a password alone insufficient for compromise.

Reference: NIST SP 800-63B — Digital Identity Guidelines

VULN-003 - Broken Authentication

CVSS

CVSS:9.9/AV:N/AC:L/PR:L/UI:N/S:C/C:G/I:G/A:L Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | High	Probability | High	Severity | High	Fix Effort | Medium

DESCRIPTION

Broken Authentication refers to flaws in the design or implementation of authentication and session-management mechanisms that allow attackers to assume other users' identities, bypass login controls, or escalate privileges. Typical failures include accepting client-supplied role/session values, predictable or insecure session identifiers, missing session invalidation, insufficient protection of credential-reset flows, and failure to require or correctly implement multi-factor authentication. These weaknesses permit attackers to impersonate legitimate users and perform actions on their behalf. OWASP Foundation+1

Broken Authentication manifests in several common forms:

- **Credential compromise and stuffing** — use of leaked username/password pairs against the application when the site allows reuse or fails to detect credential-stuffing patterns. OWASP Foundation
- **Weak or guessable credentials / online guessing** — permitting short, simple or otherwise low-entropy passwords and failing to rate-limit login attempts makes online brute-force feasible. OWASP Foundation+1
- **Session management weaknesses** — storing authoritative role or session state in client-modifiable values (plain cookies), predictable session IDs, or failing to rotate/invalidate session tokens can lead to session hijacking and privilege escalation. NIST Pages+1
- **Missing server-side authorization checks** — trusting client-supplied markers of authorization (for example, an unverified role=admin cookie) without server validation enables elevation of privilege even after a basic account compromise. cwe.mitre.org+1

The real-world impact ranges from takeover of individual low-privilege accounts to full administrative compromise, data exfiltration, fraudulent transactions, or lateral movement across systems — severity depends on which accounts can be compromised and whether authorization boundaries are enforced server-side. Attack feasibility increases when multiple enabling factors coincide: leaked credentials, absence of rate-limiting, client-side disclosure of session/role schemes, and lack of MFA.

PROOF OF CONCEPT

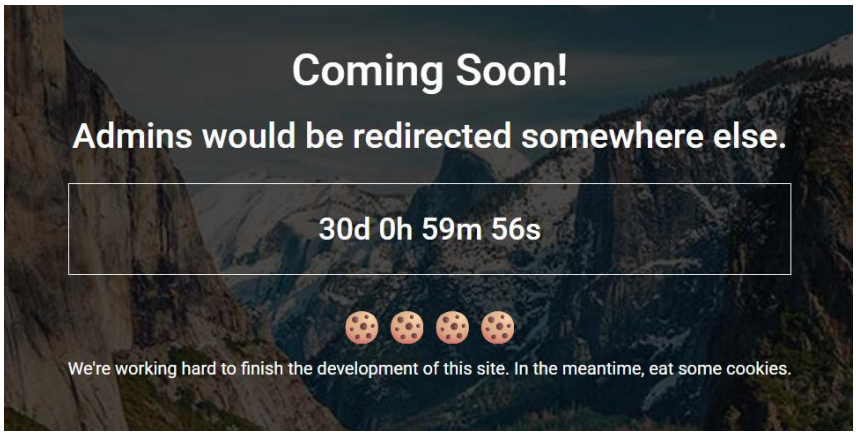


Figure 1 - Admin-only page displayed after modifying the session cookie, illustrating the Broken Authentication vulnerability.

```
Pretty  Raw  Hex
1  GET /admin/mpdf.php ?user=admin HTTP/1.1
2  Host : techie-world.xyz
3  Cookie : user=admin
4  Cache-Control : max-age=0
5  Sec-Ch-Ua : "Not;A=Brand";v="99", "Chromium";v="106"
6  Sec-Ch-Ua-Mobile : ?0
7  Sec-Ch-Ua-Platform : "Windows"
8  Upgrade-Insecure-Requests : 1
9  User-Agent : Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/106.0.5249.62 Safari/537.36
10 Accept :
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
11 Sec-Fetch-Site : same-origin
12 Sec-Fetch-Mode : navigate
13 Sec-Fetch-User : ?1
14 Sec-Fetch-Dest : document
15 Referer : https://techie-world.xyz/login/
16 Accept-Encoding : gzip, deflate
17 Accept-Language : en-US,en;q=0.9
18 Connection : close
```

Figure 2 - HTTP request showing the session cookie modified to admin, demonstrating exploitation of the Broken Authentication vulnerability.

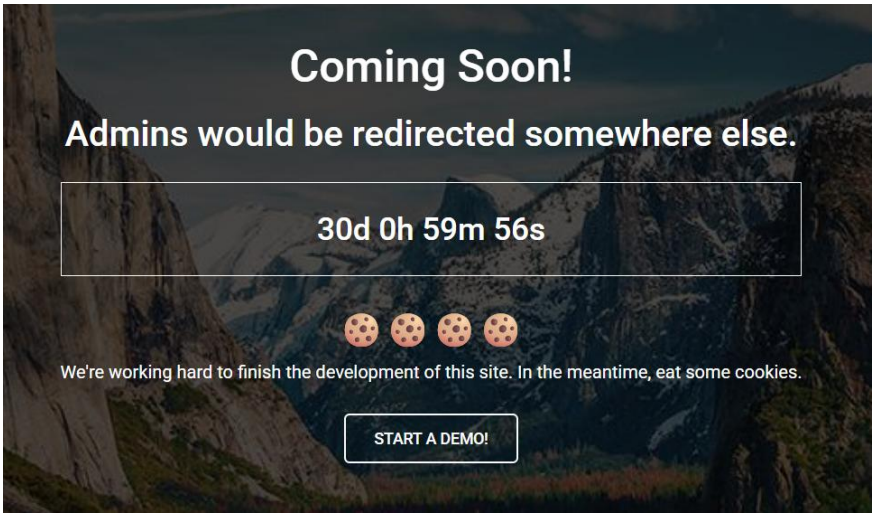


Figure 3 - Attacker-controlled landing page displayed after successful privilege escalation, illustrating that the application serves admin-only content following client-side session/role manipulation

DETAILS

I authenticated to the application using a low-privilege test account (user). After login I observed a hint indicating that additional, more sensitive content required administrator privileges. Inspecting the client-side cookies, I identified a cookie containing the role/session value (for example user=...).

I modified the cookie value in the browser from user=user to user=admin and then re-requested the previously restricted resource. The server responded by returning the admin-only content without requiring re-authentication or returning an authorization error. In other words, changing the client-side cookie from user to admin granted administrative access.

After modifying the cookie value from user to admin, the application responded by displaying the previously restricted administrative content. The transition occurred seamlessly without any further checks or prompts, which indicated that the system was relying on the client-supplied cookie value to determine access rights. This behavior confirmed that changing the cookie was sufficient to elevate privileges from a regular user session to an administrative level.

RECOMMENDED MITIGATIONS

1. **Server-Side Session Validation**
   Never rely on client-supplied values (cookies, hidden fields, query strings) for authentication or role checks. All role and privilege decisions must be verified against a secure server-side session store.
   Reference: OWASP Authentication Cheat Sheet

2. **Secure Session Management**
   Use strong, random, and unpredictable session IDs. Regenerate (rotate) session IDs after login and privilege changes, enforce expiration and timeout, and ensure sessions are invalidated on logout.
   Reference: OWASP Session Management Cheat Sheet

3. **HttpOnly, Secure, and SameSite Cookie Flags**
   Set session cookies with HttpOnly (not accessible to JavaScript), Secure (only sent over HTTPS), and SameSite (mitigates CSRF) attributes. This reduces the risk of session theft or manipulation.
   Reference:
   https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html?utm_source=chatgpt.com

4. **Re-Authentication for Privilege Escalation**
   Require users to re-authenticate (e.g., enter their password again or use MFA) before accessing sensitive functions such as administrative dashboards or account configuration changes.
   Reference:
   https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html

VULN-004 - Arbitrary File Reading

CVSS

CVSS:7.6/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:L/A:L Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | High	Probability | High	Severity | Medium	Fix Effort | Low

DESCRIPTION

Arbitrary File Read is a type of security vulnerability in web applications that allows attackers to read arbitrary files on the server's filesystem. By manipulating input parameters (for example, file names or paths), an attacker can gain unauthorized access to sensitive files such as configuration files, credentials, system files, or logs. This leads primarily to a confidentiality impact, since sensitive information may be disclosed and later abused for further exploitation.

The attack vector can be divided into several common forms, such as:

- **Path Traversal** – Using directory traversal sequences like ../ to access files outside the intended directory (e.g., /etc/passwd).
- **Local File Inclusion (LFI)** – Application code directly includes or reads files based on user input, allowing disclosure of local files.
- **Remote File Inclusion (RFI)** – The application fetches and includes remote resources (e.g., via a URL), which can lead to remote code execution if enabled.

PROOF OF CONCEPT

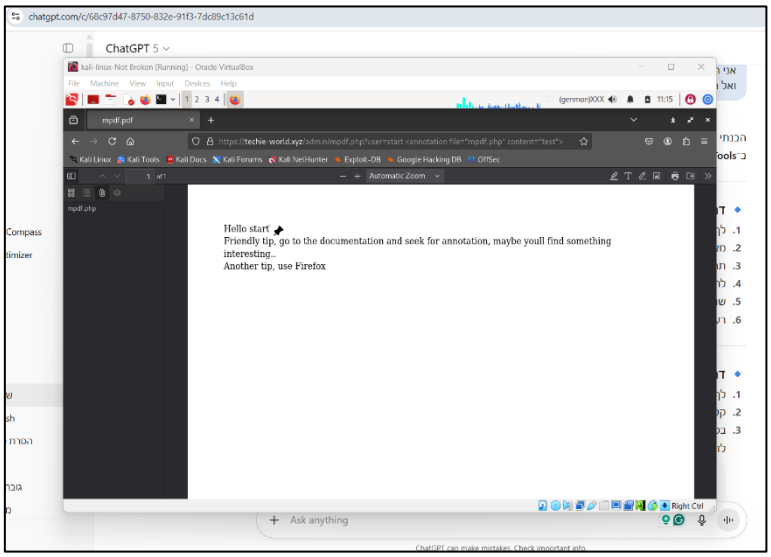


Figure 1 - Demonstration of Arbitrary File Read via the annotation parameter returning an internal file

DETAILS

I injected a crafted value into the user parameter of the /admin/mpdf.php endpoint by placing the annotation payload directly in the URL (...?user=start <annotation file="mpdf.php" content="test">).

I loaded the modified URL in a browser from a Kali Linux test machine and observed that the server returned a PDF containing the embedded annotation from mpdf.php; the request URL and returned PDF content are visible in the attached screenshot.

I captured evidence by saving screenshots of the browser showing the full address bar and the rendered PDF. No attempts were made to modify server state or execute code — testing was limited to retrieving and documenting the returned content.

RECOMMENDED MITIGATIONS

1. Strict Whitelisting of Files

Do **not** use raw user-supplied filenames/paths. Expose only logical identifiers (e.g. report_id=123) that are mapped server-side to an approved list of physical files/paths. Reject any request that isn't in the allow-list. This is the single most effective mitigation because it removes the attacker's ability to control the filesystem API.
Reference: https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.1-Testing_for_Local_File_Inclusion/

2. Move & protect sensitive files

Store configuration, .env, private keys and logs outside the webroot and restrict filesystem permissions so the webserver user cannot read them. Even if an arbitrary-read bug exists, this limits the impact by ensuring critical secrets are inaccessible.
Reference: https://www.invicti.com/web-vulnerability-scanner/vulnerabilities/local-file-inclusion/

https://owasp.org/www-community/attacks/Path_Traversal/

VULN-005 - Path Traversal

CVSS

CVSS:7.5/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | High	Probability | High	Severity | High	Fix Effort | Low

DESCRIPTION

Path traversal (also called directory traversal) is a class of web-application vulnerability that allows an attacker to manipulate file path inputs so the application accesses files and directories outside the intended file system location. By exploiting insufficient validation or improper canonicalization of user-supplied paths, an attacker can read (and in some cases include or execute) arbitrary files on the server. The impact ranges from information disclosure (configuration files, secrets, user data) to escalations that enable further attacks, depending on which files are accessible and what other protections the system has in place.

Path traversal attacks appear in several practical flavors:

• Directory traversal (simple ../ style traversal) — The attacker injects sequences that traverse up the directory tree (e.g., ../../) so the file-access routine resolves a path outside the intended base directory. If the application directly trusts and uses the supplied path, this yields arbitrary local file reads.

• Local File Inclusion (LFI) — A special case where an application includes or loads files (for example, into a template or PDF generator) based on user input. An LFI often leads to disclosure of local files and may enable remote code execution if the attacker can control file contents or include special wrappers (logs, uploads, or protocol wrappers).

• Remote File Inclusion (RFI) — Where the application fetches and uses a remote resource specified by user input. If allowed, RFI lets an attacker cause the server to retrieve and execute attacker-controlled remote content — a high-impact vector when allowed.

• Encoding and bypass techniques — Attackers often use URL encoding, double-encoding, Unicode/UTF-8 encodings, or symbolic links to bypass naive filters and reach files outside the permitted area. Robust defenses must canonicalize and normalize inputs before checks.

The severity of a path traversal vulnerability depends on what files are reachable (e.g., /etc/passwd, application configuration, .env files, SSH keys, or database credentials) and whether the server can be induced to execute attacker-controlled content. Proper mitigations (allowlisting, resolving paths with realpath/canonicalization and verifying base-directory prefixes, running file-access

code with least privilege, and avoiding direct use of user-supplied filesystem paths) dramatically reduce risk.

PROOF OF CONCEPT

Figure 1 - browser address bar showing encoded path-traversal payload

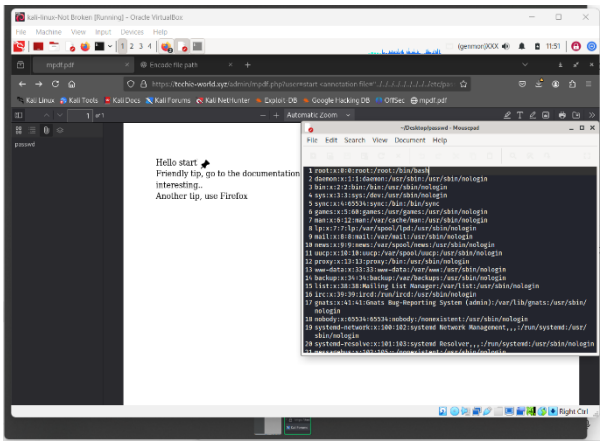


Figure 2 - PDF viewer with injected annotation (left) and terminal showing /etc/passwd output (right)

DETAILS

During testing I injected a crafted annotation value into the user parameter of the /admin/mpdf.php endpoint. The payload included directory traversal sequences (../) inside the annotation definition, pointing to the system file /etc/passwd. After submitting the request, the server returned a 200 OK response and the body contained lines directly extracted from the file, such as root:x:0:0:root:/root:/bin/bash. This confirmed that the application was processing user input without any validation and exposing local files through path traversal.

The behavior showed that the application relies on client-supplied input for building file paths without canonicalization or verification. Since no restrictions or allowlists were applied, the traversal sequences successfully escaped the intended directory and accessed sensitive files on the server. The response was immediate, and no authentication or additional interaction was required to trigger the vulnerability.

The observed impact is the disclosure of sensitive information stored in system files, configuration files, and potentially other critical resources. In this case, accessing /etc/passwd demonstrated that arbitrary local file read is possible. Such access represents a high confidentiality risk, as it could lead to exposure of credentials, keys, or application secrets, while no integrity or availability impact was observed in this test.

RECOMMENDED MITIGATIONS

1. Use Whitelists (Allow-list) Instead of Accepting Arbitrary Paths

Don't let the client supply full file paths. Instead, let them provide an identifier (ID or token) which the server maps to a safe, predefined file or directory. Reject any input that's not in the allow-list.
Reference: https://owasp.org/www-community/attacks/Path_Traversal/

2. Canonicalize Paths & Verify Against Base Directory

After concatenating base directory and user input, call a canonicalization / real-path API (e.g. realpath(), getCanonicalPath()) and check that the final resolved path starts with your intended base directory. If it doesn't, reject the request.
Reference: https://brightsec.com/blog/directory-traversal-mitigation/

3. Run File Access Under Least Privilege & Use Strong OS Controls

Even if traversal gets through, its damage should be limited. Run the file-accessing component with minimal permissions (only read access to allowed directories), disable symlink following when possible, use containerization or chroot jails, and harden OS/file system permissions so that even if someone escapes the base directory path, they can't access sensitive files.
Reference: https://brightsec.com/blog/directory-traversal-mitigation/

https://www.invicti.com/learn/directory-traversal-path-traversal/

VULN-006 – Remote Code Execution

CVSS

CVSS:9.8/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | High	Probability | High	Severity | High	Fix Effort | Medium

DESCRIPTION

Remote Code Execution (RCE) is a class of critical security vulnerability that allows an attacker to execute arbitrary code on a remote host or service. When an application accepts untrusted input and passes it (directly or indirectly) to an interpreter, OS shell, runtime or system API without adequate validation or isolation, an attacker can craft input that causes the target to run attacker-controlled commands or programs. RCE vulnerabilities vary in severity from local privilege escalation to full system compromise, depending on the context and the privileges of the process that executes the code.

The RCE attack vector can be divided into several common methodologies and root causes, such as:

● **Command injection** – Occurs when user-supplied input is concatenated into shell commands (for example via system(), exec() or similar APIs) and not properly validated or escaped. If the application runs those commands with elevated privileges, an attacker can run arbitrary OS commands.

● **Unsafe deserialization** – When applications deserialize objects from untrusted sources (JSON, PHP serialized objects, Java/.NET deserialization), specially crafted objects can trigger execution of gadget chains in application libraries, leading to arbitrary code execution inside the application runtime.

● **Template / server-side injection** – Server-side template engines (e.g., when templates permit evaluation of expressions) or server-side scripting that evaluates user input can be abused to run code in the context of the rendering process.

● **File upload / file include vulnerabilities** – Allowing users to upload files that are later executed, or including files based on attacker-controlled paths (e.g., insecure file inclusion), can result in the server executing attacker-controlled code (uploading a script and then visiting it, or causing the service to include it).

● **Third-party component exploitation** – Vulnerable libraries, plugins or services that accept input which can then trigger code execution (for example, image/PDF libraries that offer features to run callbacks or load external resources) can be an RCE source if not hardened.

● **Destructive server features / unsafe eval** – Use of generic eval()/exec() on user-supplied strings, or other dynamic evaluation features, enables RCE when the input is not strictly controlled.

PROOF OF CONCEPT

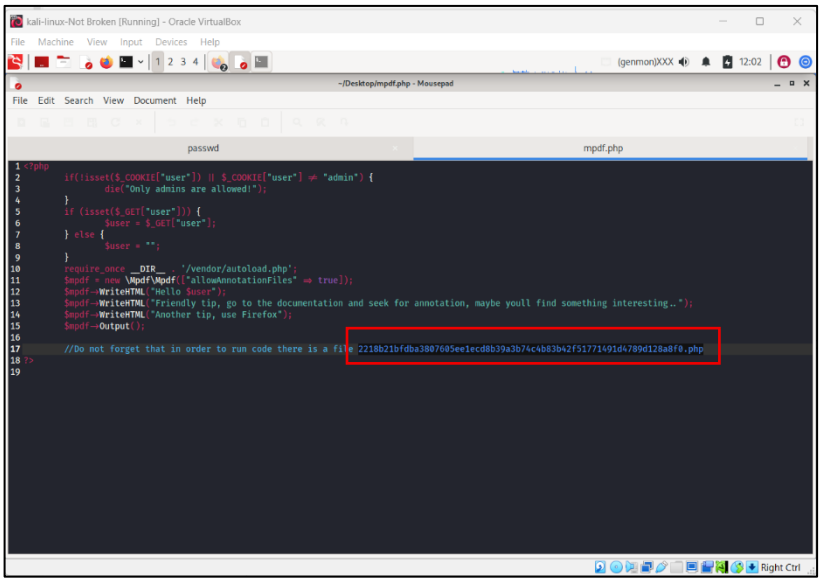


Figure 1 - Source code snippet from mpdf.php highlighting the attacker-controlled execution file hash 2218b21bfdb...a8f0.php (RCE proof-of-access demonstrated)

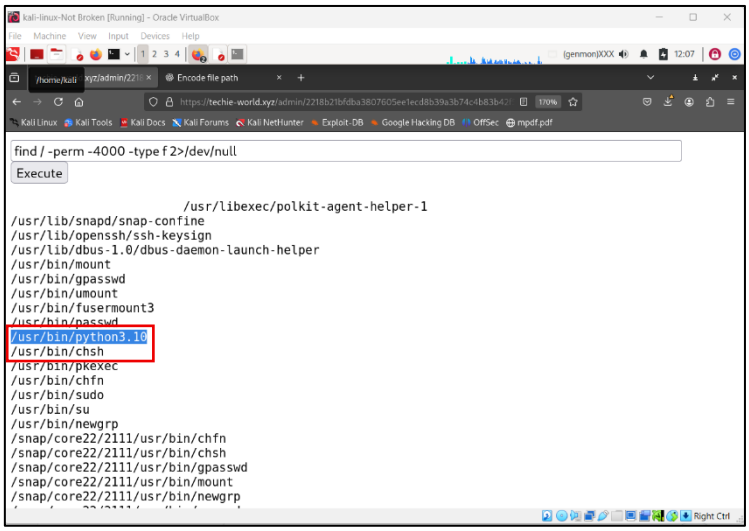


Figure 2 - Highlighted interpreter path /usr/bin/python3.10; supplying the attacker file hash produced a command shell (proof-of-access)

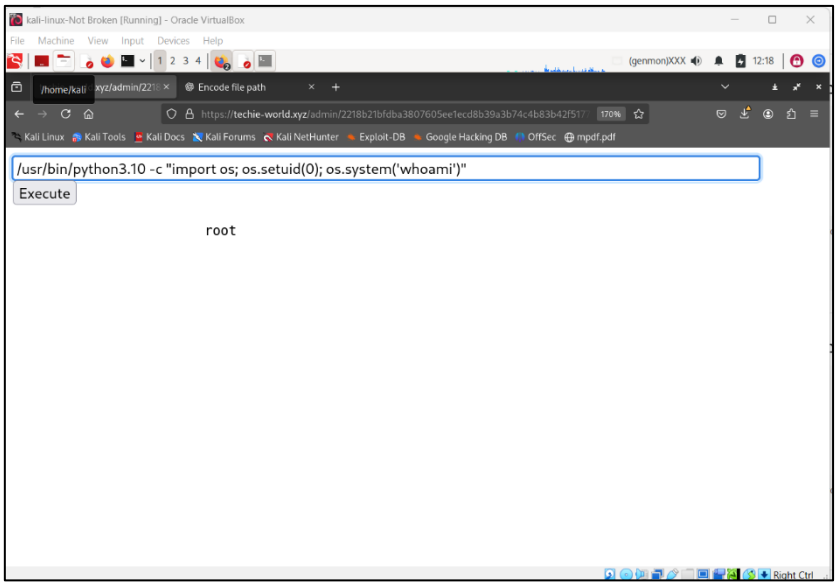


Figure 3 - command output showing whoami returned root after executing the attacker-controlled file/hash

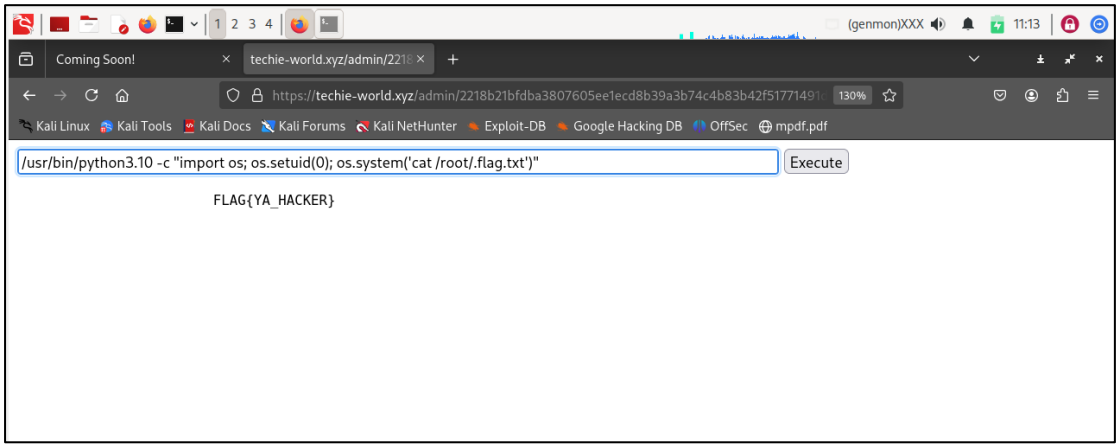


Figure 4 - executing attacker-controlled command returned sensitive file contents FLAG{YA_HACKER}

DETAILS

During testing I injected the identified annotation **hash** into the user parameter of /admin/mpdf.php. Supplying that exact hash caused the application to resolve and load the attacker-controlled annotation file (e.g. …/annotations/2218b21bfdb…a8f0.php), and the server returned an interactive command-output context. From that session I executed find / -perm -4000 -type f 2>/dev/null to enumerate SUID files; the command returned entries including /usr/libexec/polkit-agent-helper-1 and /usr/bin/python3.10. I then invoked the Python interpreter with an inline script: /usr/bin/python3.10 -c "import os; os.setuid(0); os.system('whoami')", which returned root. Finally, after elevating privileges in that process, I executed /usr/bin/python3.10 -c "import os; os.setuid(0); os.system('cat /root/.flag.txt')" and retrieved the flag contents.

Technically, the hash functioned as the execution entry point because the application used the client-supplied token directly when building the filesystem path; injecting the hash made the server locate and load the attacker file. The find /

-perm -4000 -type f 2>/dev/null command scans the whole filesystem (find /) for regular files (-type f) with the SUID bit set (-perm -4000) and silences errors (2>/dev/null) so only matches are shown; SUID files run with the file owner's effective UID and are therefore common privilege-escalation candidates. /usr/libexec/polkit-agent-helper-1 appearing in the results indicates the presence of SUID helper binaries on the host; /usr/bin/python3.10 in the results is significant because an interpreter with SUID-capable permissions can be used to obtain elevated capabilities. Invoking /usr/bin/python3.10 -c "import os; os.setuid(0); os.system('whoami')" starts the interpreter, calls os.setuid(0) to set the process UID to 0 (root) when the process context allows it, and runs whoami via os.system() — the whoami output root therefore demonstrates the process successfully changed to UID 0. Repeating the sequence with os.system('cat /root/.flag.txt') executed cat under the root context and returned the file contents, proving read access to a root-owned file.

Evidence: screenshots/logs attached show (1) the mpdf.php code with the annotated hash (execution entry point), (2) the find / -perm -4000 -type f 2>/dev/null output including /usr/libexec/polkit-agent-helper-1 and /usr/bin/python3.10, (3) the whoami output showing root, and (4) the cat /root/.flag.txt output containing the flag.

RECOMMENDED MITIGATIONS

1. Attacker-controlled hash used as a filename (execution entry point)

Do not let a client-supplied hash/token map directly to a filesystem path. Accept only a short identifier and perform a server-side lookup (token → pre-approved filename). After composing the server path always canonicalize it (realpath()/getCanonicalPath()), verify the resolved path begins with the intended base directory, and reject requests containing traversal (..), null bytes or absolute paths; maintain an allow-list of valid annotation IDs in a server-side registry so arbitrary strings cannot cause file loads.
Reference: https://owasp.org/www-community/attacks/Path_Traversal
https://portswigger.net/web-security/file-path-traversal

2. SUID interpreter present on the host (privilege escalation vector)

Audit and remove unnecessary SUID/SGID bits from interpreters and helper binaries (e.g. Python). Run file-processing services as dedicated non-root users, make upload directories non-executable, and remove SUID from binaries that do not require it (chmod u-s /path/to/binary) or replace with capability-based controls. This prevents the "invoke SUID interpreter → setuid(0)" escalation path you observed.
Reference: https://access.redhat.com/solutions/33374/

3. Executing user-supplied content on the host (lack of isolation)

Never execute untrusted user content on the host. If processing is required, run it inside a strict ephemeral sandbox (container/VM) that does **not** mount host-sensitive paths, runs processes as non-root, applies syscall filters (seccomp) and resource limits, and disallows access to host interpreters. Proper isolation ensures a loaded attacker file cannot reach host SUID binaries or /root even if application logic is vulnerable.

Reference:
https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html/

VULN-007 - Cross Site Scripting (Reflected XSS)

CVSS

CVSS:5.4/AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:L/A:N Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | Medium	Probability | High	Severity | Medium	Fix Effort | Low

DESCRIPTION

Cross-site scripting (XSS) is a type of security vulnerability typically found in web applications. XSS attacks enable attackers to inject client-side scripts (JavaScript and HTML content) into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy. XSS effects vary in range from petty nuisance to significant security risk, depending on the sensitivity of the data handled by the vulnerable site and the nature of any security mitigation implemented by the site's owner network.

The XSS attack vector can be divided into several different methodologies, such as:

- Reflected XSS – Occurs when a server reflecting a parameter sent from the client into the content of the page itself, without properly sanitizing the content.

- Stored XSS – Occurs when the data provided by the attacker is saved by the server, and then permanently displayed on "normal" pages returned to other users in the course of regular browsing, without proper HTML escaping.

- DOM-based XSS - As the JavaScript code was also processing user input and rendering it in the web page content, a new sub-class of reflected XSS attacks started to appear that was called DOM-based cross-site scripting. In a DOM-based XSS attack, the malicious data does not touch the web server. Rather, it is being reflected by the JavaScript code, fully on the client side.

PROOF OF CONCEPT

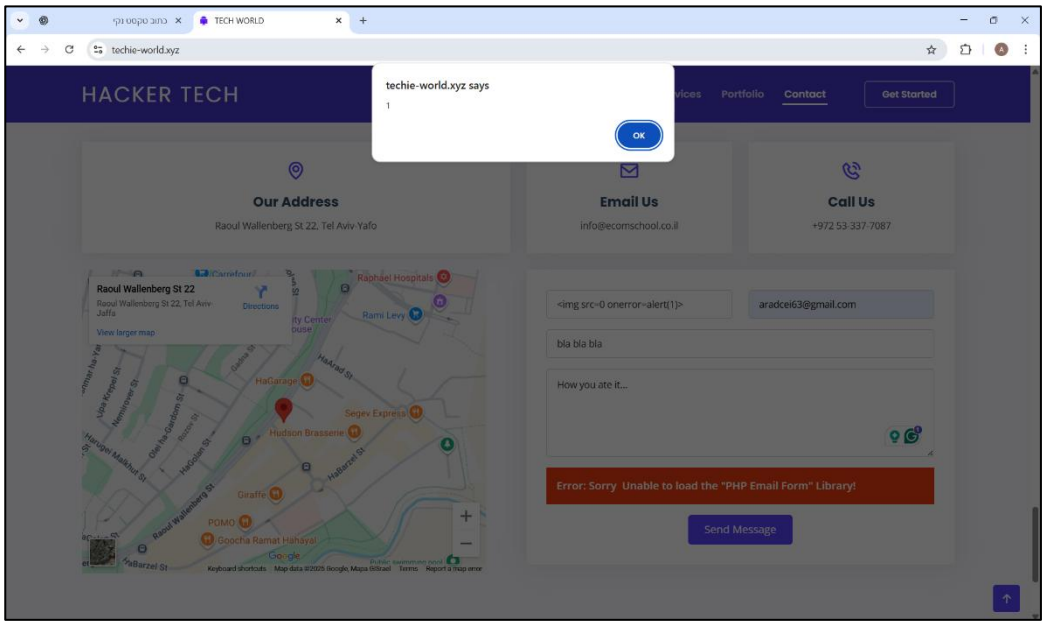


Figure 1 - Reflected XSS payload execution in the contact form

DETAILS

During testing I discovered a reflected cross-site scripting (XSS) vulnerability in the **name** field of the contact form. I submitted the payload <img src=0 onerror=alert(1)> in the name input and then sent the message. The application reflected the exact input back into the page without proper encoding, and the browser executed the injected JavaScript, causing an alert dialog to appear. This confirms a reflected XSS condition where attacker-supplied content is returned and executed in the victim's browser.

Because the payload is reflected (not stored), exploitation requires a user to open a crafted link or submit the malicious input (user interaction is required). The vulnerability allows arbitrary script execution in the context of the site and could be abused for UI redress, phishing, or limited data exposure (depending on other protections such as HttpOnly cookies and CSP). Remediation should focus on context-aware output encoding for the name field and additional defenses (see mitigations).

RECOMMENDED MITIGATIONS

1. Context-aware output encoding / escaping

Always encode user-supplied data at output according to the exact rendering context (HTML body, HTML attribute, JavaScript, CSS or URL). Do not rely on input filtering alone — proper context-aware escaping is the primary defense against XSS.

Reference: cheatsheetseries.owasp.org

2. Server-side allowlist (regex) + sanitizer (defense-in-depth)

Validate inputs on the server using allowlist patterns (regex) that strictly match expected values (e.g. name fields), reject anything that does not conform, and—if any HTML must be allowed—sanitize it with a vetted library (e.g. DOMPurify). Regex-based validation is useful but **not** sufficient by itself, so combine it with

output encoding or a sanitizer.

Reference:

https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html?utm_source=chatgpt.com#regular-expressions-regex

VULN-008 - Cross-Site Request Forgery

CVSS

CVSS:5.3/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | Medium	Probability | High	Severity | Low	Fix Effort | Low

DESCRIPTION

Cross-Site Request Forgery (CSRF) is a web security vulnerability that forces an authenticated user's browser to execute unwanted actions on a web application. Because browsers automatically include authentication information such as cookies or session tokens with each request, an attacker can craft a malicious link or hidden form that causes the victim's browser to send a forged request to the target application.

If the application does not properly verify the legitimacy of the request, the attacker can exploit the user's active session to perform actions such as changing account settings, transferring funds, or escalating privileges. The impact depends on the level of access held by the targeted user, with attacks against administrative accounts posing the highest risk.

PROOF OF CONCEPT

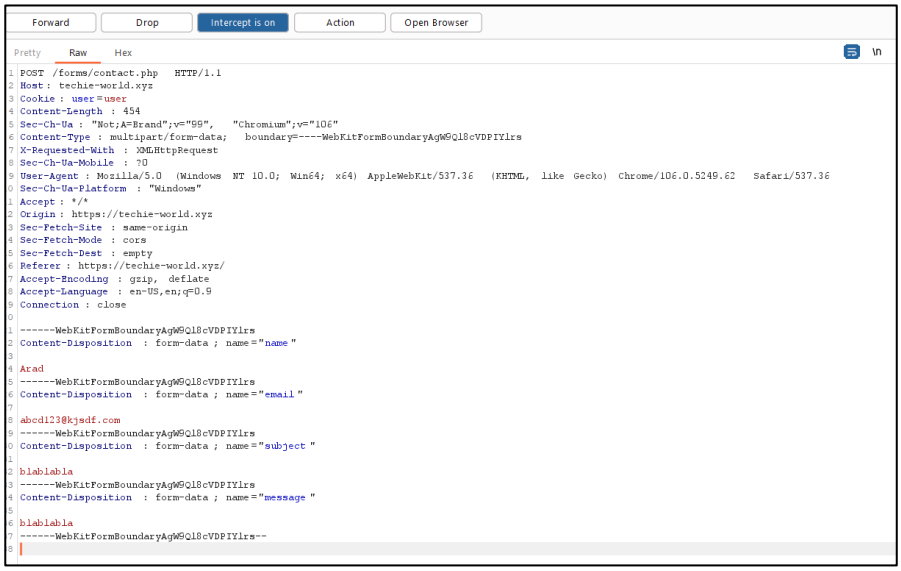


Figure 1 - Example of a CSRF attack triggered from an attacker-controlled page

Figure 2 - Example of a CSRF request sent to /forms/contact.php, showing form fields (name, email, subject, message) and session cookie automatically included by the browser

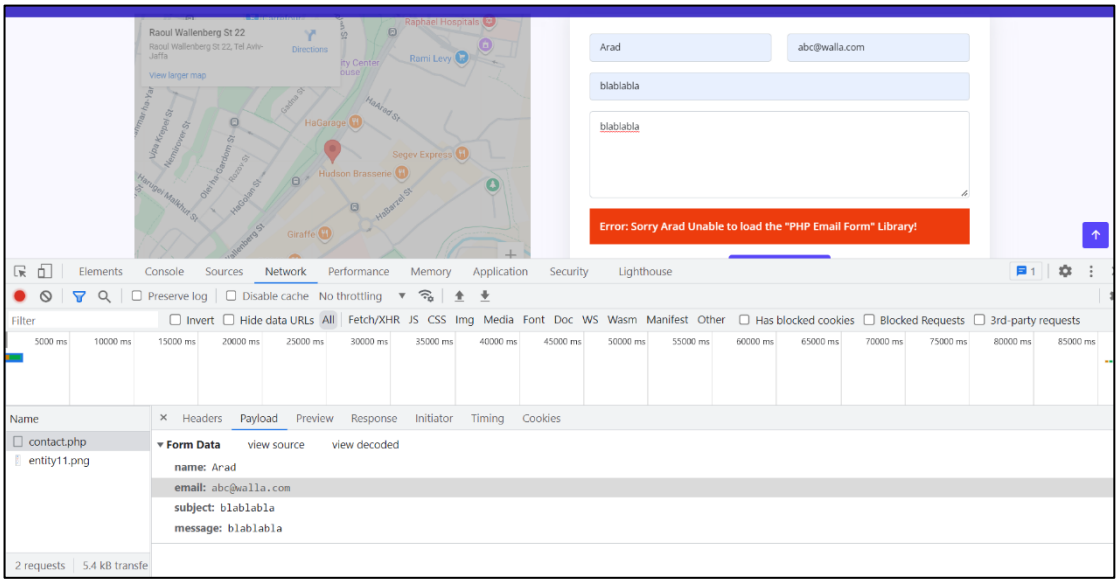


Figure 3 - Network panel showing POST request to contact.php with captured form data (name, email, subject, message), triggered from the attacker-controlled page

DETAILS

During testing a normal **POST request** was submitted to the contact form endpoint (/forms/contact.php) with parameters such as name, email, subject, and message. Using the browser's developer tools (Network panel), it was observed that the request was transmitted together with the user's active session cookie (user=user). The form data appeared in the raw request payload and was accepted by the server without additional validation.

The key finding is that the application processes these requests even when they originate from external sources, and no anti-CSRF defenses were detected (such as CSRF tokens, SameSite cookie attributes, or strict Origin/Referer validation). This behavior confirms that any third-party page could cause a logged-in user's browser to automatically submit a forged request carrying their valid session credentials.

Although the test only involved sending legitimate form values, the implication is that an attacker could replicate this behavior and trick authenticated users into unknowingly submitting requests. Depending on the server-side functionality of the vulnerable endpoint, this could be leveraged to perform unauthorized actions on behalf of users without their consent.

RECOMMENDED MITIGATIONS

1. CSRF Tokens (Synchronizer Token Pattern)

Generate a cryptographically secure anti-CSRF token on the server for each state-changing form and validate it on submission. Reject requests missing or containing invalid tokens.

Reference: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

2. Origin / Referer Header Validation

For sensitive requests, verify that the Origin or Referer header matches the application's trusted domain(s). Block requests with missing or mismatched headers. This provides an effective, simple check against cross-site submissions when implemented correctly.

Reference: https://developer.mozilla.org/en-US/docs/Web/Security/Practical_implementation_guides/CSRF_prevention

3. Enable Framework / Built-in CSRF Protections

Use the proven CSRF protection middleware provided by your web framework (do not roll your own unless absolutely necessary). Frameworks typically implement token generation, validation, and integration with templating systems. Enable and configure these features according to the framework docs.

Reference: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

VULN-009 – HTML Injection

CVSS

CVSS:5.3/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N Calculated by https://www.first.org/cvss/calculator/3.1

RISK

General | Medium	Probability | High	Severity | Medium	Fix Effort | Low

DESCRIPTION

HTML Injection is a vulnerability where an attacker can inject arbitrary HTML markup into a web page that will be rendered by other users' browsers. This occurs when user-supplied input is included in a page's HTML output without proper validation or encoding, allowing an attacker to modify page structure, insert misleading content, or alter the presentation and behavior of the interface. Injected HTML can be used to display fake forms or messages, deface pages, or manipulate the DOM in ways that influence user interactions and expectations.

The security impact of HTML Injection ranges from low-level visual tampering to higher-risk scenarios when combined with other weaknesses. While pure HTML injection (inserting tags or attributes only) primarily affects appearance and user trust, it can facilitate phishing, spoofed UI elements, or the delivery of payloads that lead to script execution when the application also permits scriptable contexts or fails to sanitize attributes (e.g., href, src, on*). Preventing HTML Injection requires output encoding appropriate to the HTML context, strict input validation, and avoiding direct insertion of untrusted content into page templates.

PROOF OF CONCEPT

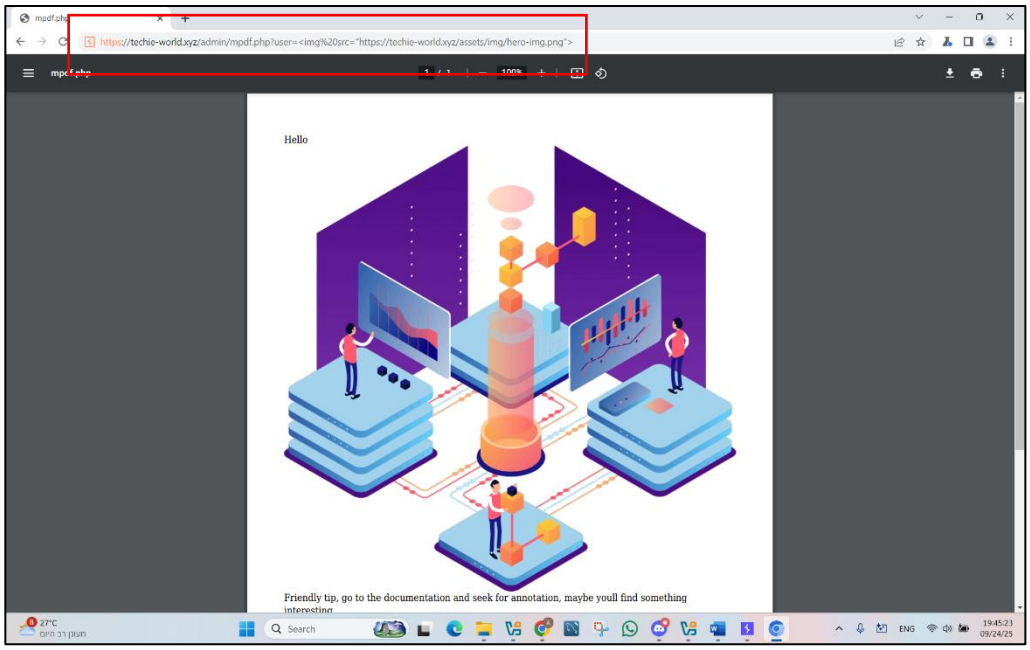


Figure 1 - Attacker-controlled URL injecting HTML into mpdf rendering (embedded <img> in user parameter)

DETAILS

During testing it was possible to inject raw HTML into the user parameter passed to mpdf.php; specifically, an <img src="..."> tag was submitted and the image markup was rendered inside the PDF output. The captured screenshot shows the injected tag in the address bar and the PDF renderer displaying content that was derived from the attacker-supplied input, which demonstrates that the application inserts user input into HTML output without proper encoding or sanitization. This is a clear case of HTML Injection (in this instance used to embed an image), and it confirms that untrusted data flows directly into rendered HTML/PDF contexts.

The practical implications range from straightforward visual tampering to more serious security issues when combined with other weaknesses. At minimum, an attacker can inject misleading content, fake UI elements, or deface pages/PDFs to deceive users. If the application exposes scriptable contexts or fails to sanitize attributes, the same weakness may enable attribute-based attacks or lead to XSS (script execution) in other contexts. HTML injection can also facilitate social-engineering (phishing) and can be chained with other flaws (CSRF, file inclusion, or RCE) to increase impact, so addressing output encoding and input validation at the rendering boundary is essential.

RECOMMENDED MITIGATIONS

1. Contextual Output Encoding

Always encode user-supplied input according to the HTML context where it is inserted. Use HTML encoding for text nodes (e.g., in PHP: htmlspecialchars($input, ENT_QUOTES, 'UTF-8')), attribute encoding for attributes, and URL encoding with scheme validation for links. This ensures that input is rendered as text only and not interpreted as HTML markup.

Reference:

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html?utm_source=chatgpt.com

2. Input Validation & Strict Handling of Untrusted Data

Apply strict input validation rules to define exactly what values are allowed in each field (e.g., alphanumeric only, valid email format). Reject or sanitize all unexpected input. For fields rendered in PDF/HTML, ensure only plain text is permitted—disallow tags or attributes entirely. This blocks attempts to inject malicious HTML before it reaches the rendering stage.

Reference:

https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

APPENDICES

METHODOLOGY

The work methodology of our penetration testing team includes some of the following potential inspected information according to the client's needs:

APPLICATIVE PENETRATION TESTS

The test was conducted identify the following:

- Vulnerable functions used in the code.
- Un-sanitized Input provided by the user.
- Well known vulnerabilities exists in the system.
- Sensitive information leakage.

Performed general inspection of the code if requested by the client. In addition to the usage of automated tools to identify vulnerabilities and potential issues in the target application.

Understanding the system logic – Before performing the test, the testers watched and examined the system in order to understand its purpose and mode of operation. During this exam the examiners try to understand the following:

- **Client Requests**:
  - Examined hidden parameters.
  - Examine important parameters that are in outgoing requests
  - Examine paths and form of loading of data on the site
- **Server Answers:**
  - Check when a cookie is created or when the content of the cookie changes.
  - Examine the number of errors that recur from the site.
  - Examine when the server returns redirection in order to find *Open Redirect*.

- **Understanding the customer side of the system:**
    - The testers examined which JavaScript functions are called in the code.
    - Examined whether HTML code can be injected next to a client.
    - Examined whether Web Socket technology is used and what information passes through it.
- **Data collection and scanning:**
    - Find additional servers and get information about those servers.
    - Scans were also performed by dedicated tools in order to find known vulnerabilities on the site.
- **Checking the user's identity management and authorization:**
    - The examiners examined the permission level in the system, what permission level they are at and whether it is possible to switch to another permission level.
- **Checking the user authentication process:**
    - The testers examined the mechanism of connection to the system, whether there is Anti-Automation protection such as CAPTCHA.
- **Authentication of the resulting input:**
    - The testers examined the user's call management in addition to verifying the inputs sent from the client alongside the server. Attempts were also made and exploits of systems to upload documents to the system, file reading systems and even injecting malicious code into the system.
- **Error management in the system:**
    - During the test, errors that were repeated by a customer were identified and conclusions were drawn according to the same errors that helped the testers during this test.
- **Logical Bypasses:**
    - During the test, the testers questioned the system logic in order to check the transition between forms, switching between one user and another, making a registration in the system and more. In order to test whether non-programmed operations can be performed by default.
- **Testing of potential attack vectors, and provideing a working POC for examination.**
- **The test result is a detailed report contains all the findings details about the vulnerabilities found:**

- CVSS
- RISK
- DESCRIPTION.
- POC
- DETAILS
- RECOMMENDED MITIGATIONS

- **Additionally, the following elements may be performed due to the client's request:**
  - Conducting a re-test to the system in order to verify the security again.
  - Providing the development team from "ECOM" to support the client during the mitigation process.
  - Providing the penetration testing team from "ECOM" explain in more depth about the report.

FINDINGS CLASSIFICATIONS

The purpose of the presentation in the manner illustrated above is on several levels:

1. **The vulnerability name -** A main vulnerability of which an examination is performed.

2. **Description of the test -** Main description about the vulnerability.

3. **Findings of the test -** Findings that clearly and concisely describe an existing situation. The purpose of the section is to document the existing situation as found during the examination. The test results can be normal or in a status that endangers the entire array tested, at the level of exposure to damage to activity continuity, leaked sensitive information or damage to property and people.

4. **The risks as a result of the existing situation -** A rating that clarifies what is the risk arising to the customer from the findings.

5. **Severity of the damage -** The method of determining the level of damage is performed according to the following details:

   Critical – For the following risks:

   - The realization of the risk will lead to a horizontal impairment in the information availability of the organization's systems and / or infrastructure.
   - The realization of the risk will lead to the disclosure of information that may threaten the stability of the organization or endanger human lives.
   - Unauthorized disruption / alteration of information that may threaten the stability of the organization or endanger human life.

   High - For the following risks:

   - The realization of the risk will impair the information availability of a sensitive system.
   - Exposure of sensitive information.
   - Unauthorized disruption / change of sensitive information in the system.

   Medium - For the following risks:

   - The realization of the risk will lead to the immediate and direct shutdown of an insensitive system.

- The realization of the risk may, in an uncertain manner, lead to the shutdown of a sensitive system.
- Exposure of non-public inside information.
- Unauthorized disruption / change of information that is not sensitive in the system in a way that will require a lot of effort in data recovery.

Low - For other serious risks.

Informative - For information provided.

6. Probability of realization - how to define the reasonableness of the risk:

Critical - A critical likelihood will be defined in a situation where it is found that the exposure has already been actually exercised (by a non-examining entity) or is available for immediate exploitation without the need for any preparation.

High - High probability will be defined in the following situations:

- The risk can be realized by Social Engineering simply.
- No technological knowledge is required or the required technological knowledge is not extensive.
- Well-documented behavior.
- The time required to realize the risk is small.
- Ability to use mechanized tools.

Medium - Moderate likelihood will be defined in the following situations:

- Information is available online.
- Well-documented behavior.
- The period of time required to realize the risk is long.

Low - Lower than moderate probability or in situations only theoretically there is a chance of exploiting the weakness.