VA/PT REPORT

VULNERABILITY ASSESSMENT & PENETRATION TEST

PREPARED FOR:

Client Organization / Client Organization

Target Scope: http://testphp.vulnweb.com/

PREPARED BY:

Cyber Advisory LLC

ExploitFinder Security Team

DOCUMENT ID: `ee71f143-f81d-4d97-a022-d2d07c93be0e`

DATE: 2026-02-08 00:49:06

STRICTLY CONFIDENTIAL

This document contains confidential information regarding the security posture of the target system. Distribution is restricted to authorized personnel only.

TABLE OF CONTENTS

1. DISCLAIMER & CONFIDENTIALITY

This report is the exclusive property of the Client and Cyber Advisory LLC. The content of this document is strictly confidential and intended solely for the use of the individual or entity to whom it is addressed.

LIMITATION OF LIABILITY:
This assessment was performed using industry-standard methodologies (NIST, OWASP, OSSTMM) and the advanced ExploitFinder engine. While every effort has been made to ensure accuracy, the security landscape is continuously evolving. This report represents a snapshot of the security posture at the time of testing. Cyber Advisory LLC cannot guarantee that all vulnerabilities have been identified, nor can it guarantee immunity from future attacks.

Cyber Advisory LLC shall not be held liable for any damages, direct or indirect, arising from the use or misuse of the information contained within this report.

2. DOCUMENT CONTROL

Role	Name	Status	Date
Lead Auditor	ExploitFinder Engine	Completed	2026-02-08 00:49:06
QA Reviewer	Cyber Advisory Team	Approved	2026-02-08 00:49:06
Report ID	ee71f143-f81d-4d97-a022-d2d07c93be0e	Version	1.0

2.B DOCUMENT VERSION HISTORY

This section tracks all revisions for audit trail and quality assurance purposes.

Version	Date	Author	Changes	Reviewed By
1.0	2026-02-08 00:49:06	ExploitFinder Engine	Initial release - Full VA/PT assessment	Cyber Advisory Team
1.1			[Reserved for future revision]	
2.0			[Reserved for future revision]	

All revisions must be approved by the QA Reviewer before distribution. Superseded versions must be destroyed or clearly marked as obsolete.

2.A ENGAGEMENT AUTHORIZATION & SCOPE

This assessment was performed under written authorization and agreed scope. Key engagement details are recorded below.

Field	Value
Engagement ID	Not Provided
Contract Reference	Not Provided
Authorized By	Not Provided
Authorization Date	Not Provided
Testing Window	Not Provided
Primary Contact	Not Provided

In-Scope Assets

Out-of-Scope Assets

Rules of Engagement

Not Provided

Assumptions

Not Provided

Data Handling

Not Provided

Limitations

Not Provided

Attestation: This report reflects technical testing within the authorized scope. It does not constitute a certification unless explicitly stated in the engagement letter and signed by authorized parties.

Client Authorized Representative: ________________________ Date: ____________

Lead Auditor: ______________________________________ Date: ____________

REGULATORY COMPLIANCE DASHBOARD

Selected Framework: NIST SP 800-53. Full multi-framework posture shown below. Maturity scores (0-5) reflect automated technical assessment only.

Framework	Status	Gaps	Maturity	Coverage	Pass	Partial	Controls
ISO 27001:2022	**TECHNICAL GAP**	15	1/5 Ini	21%	2	2	19
NIST SP 800-53	**TECHNICAL GAP**	10	1/5 Ini	29%	1	3	14
GDPR (EU)	**TECHNICAL GAP**	4	1/5 Ini	33%	1	1	6
SOC 2 Type II	**TECHNICAL GAP**	6	1/5 Ini	40%	2	2	10
HIPAA (USA)	**TECHNICAL GAP**	5	1/5 Ini	38%	1	2	8
Essential 8 (AU)	**TECHNICAL GAP**	5	1/5 Ini	38%	2	1	8
Cyber Essentials (UK)	**TECHNICAL GAP**	3	1/5 Ini	40%	0	2	5
OWASP Top 10	**TECHNICAL GAP**	6	1/5 Ini	40%	3	1	10

Maturity Scale: 1=Initial 2=Developing 3=Defined 4=Managed 5=Optimized

Note: This dashboard is an automated technical mapping based on detected vulnerabilities. It is informational only and does not constitute a certification or full compliance audit. Organizational, people, and physical controls are not assessed. Maturity scores reflect technical posture only and may differ from a full management-level assessment.

3. EXECUTIVE SUMMARY

Cyber Advisory LLC was commissioned to perform a Vulnerability Assessment and Penetration Test (VA/PT) against the infrastructure of TESTPHP.VULNWEB.COM.

The objective of this engagement was to identify security weaknesses, misconfigurations, and vulnerabilities that could be exploited by malicious actors to compromise the Confidentiality, Integrity, and Availability of the organization's assets.

Methodology Scenario:
The assessment was conducted effectively in a Black-Box Scenario. In this mode, the security team has zero prior knowledge of the target infrastructure, simulating a real-world external attack from the internet. This approach provides the most realistic view of the risk exposure to external threats.

Overall Risk Rating: CRITICAL

Critical vulnerabilities were identified with severe business impact potential. Immediate containment, emergency patching, and executive escalation are required.

Executive Risk Conclusion: CRITICAL exposure. Immediate containment and emergency remediation are required before standard business operations continue.

Summary of Results

- Executive Risk Conclusion: CRITICAL exposure. Immediate containment and emergency remediation are required before standard business operations continue.
- Report ID: ee71f143-f81d-4d97-a022-d2d07c93be0e
- Assessment date: 2026-02-08 00:49:06
- Assets analyzed: 1 IP(s), 32 subdomain(s)
- Total findings: 32 (Critical 1, High 21, Medium 3, Low 4, Info 3)

Top Finding Families

- Absence of Anti-CSRF Tokens
- Config
- Content Security Policy (CSP) Header Not Set
- Critical
- Cross Site Scripting (Reflected)
- Email Security
- GDPR Contact Missing
- GDPR Cookie Consent Missing

4. SCOPE & TECHNICAL METRICS

The following metrics summarize the depth of the assessment:

Metric	Count
IP Addresses Analyzed	1
Subdomains Enumerated	32
Vulnerabilities Identified	32

Penetration Test Scope Coverage

Penetration testing activities were executed across the authorized external attack surface: 2 reachable web assets out of 33 discovered hostnames, 0 hosts with open services, 0 validated open port-service entries, and 0 resolved public IP target(s). All in-scope subdomains, IP targets, and discovered services were fingerprinted and analyzed for exploitable weaknesses.

Network Surface Summary

Metric	Count
Discovered Hostnames	33
Reachable Assets (HTTP response observed)	2
Redirect Responses (3xx)	0
Access-Controlled / Blocked (401/403/429)	0
Dead / Unresolved	32

Network Surface Inventory (All Discovered Subdomains)

Host	HTTP Status
a105.testphp.vulnweb.com	dead
a196.testphp.vulnweb.com	dead
aomenhefabocaiwang.testphp.vulnweb.com	dead
baomahuiyulechengqipai.testphp.vulnweb.com	dead
bet365dabukailiao.testphp.vulnweb.com	dead
biboyulekaihu.testphp.vulnweb.com	dead
dalianxinyuwangqipai.testphp.vulnweb.com	dead
dubogongsi.testphp.vulnweb.com	dead
ens1.testphp.vulnweb.com	dead
hnd.testphp.vulnweb.com	dead
host-158.testphp.vulnweb.com	dead

4.N NETWORK SURFACE INVENTORY (CONTINUED)

Host	HTTP Status
jinpaiyulechengaomenduchang.testphp.vulnweb.com	dead
l33.testphp.vulnweb.com	dead
lilaizhenrenyulecheng.testphp.vulnweb.com	dead
liubowenxinshuizhuluntan.testphp.vulnweb.com	dead
liupanshui.testphp.vulnweb.com	dead
n155.testphp.vulnweb.com	dead
nico.testphp.vulnweb.com	dead
ouzhoubeizhibo.testphp.vulnweb.com	dead
phpadmin.testphp.vulnweb.com	dead
quaomenxianshangyulecheng.testphp.vulnweb.com	dead
qx7.testphp.vulnweb.com	dead
s112.testphp.vulnweb.com	dead
shalongguojibaijialeyulecheng.testphp.vulnweb.com	dead
sieb-web1.testphp.vulnweb.com	dead
srv240.testphp.vulnweb.com	dead
taianlanqiuwang.testphp.vulnweb.com	dead
testphp.vulnweb.com	200
vpn0010.testphp.vulnweb.com	dead
www.testphp.vulnweb.com	dead
xunyinglanqiubifenzhibo.testphp.vulnweb.com	dead
yulexinxiwangbocai.testphp.vulnweb.com	dead
zhenrenyulekaihu.testphp.vulnweb.com	dead

REGULATORY AUTHORITY & SCOPE

Regulatory Authority	National Institute of Standards and Technology (NIST), U.S. Department of Commerce
Legal Basis / Standard	NIST SP 800-53 Rev. 5 - Security and Privacy Controls for Information Systems and Organizations; Federal Information Security Modernization Act (FISMA)
Certification / Audit Body	FedRAMP Joint Authorization Board (JAB) / Agency Authorizing Official (AO)
Applicable Clauses	FIPS 199 (Security Categorization), FIPS 200 (Minimum Security Requirements), SP 800-37 (Risk Management Framework), SP 800-53A (Assessment Procedures)
Controls in Scope	AC (Access Control), AU (Audit), CM (Configuration), IA (Identification/Auth), IR (Incident Response), RA (Risk Assessment), SC (System/Comms), SI (System/Info Integrity)

Scope & Limitation Statement

This report provides technical evidence for NIST control family assessment. It supports the System Security Plan (SSP), Plan of Action and Milestones (POA&M), and Authorization to Operate (ATO) processes under FISMA/FedRAMP.

IMPORTANT: This VA/PT technical assessment provides supporting evidence for the regulatory framework indicated above. It does NOT replace a full management-level audit, certification, or formal assessment by an accredited body. Organizational, procedural, physical, and people controls are outside the scope of automated technical testing and must be evaluated separately.

NIST SP 800-53 Analysis

Ref: NIST - Security and Privacy Controls for Info Systems

Maps technical findings to NIST SP 800-53 Rev. 5 control families (FISMA/FedRAMP).

Automated technical mapping only. Organizational, people, and physical controls are not assessed. This is not a certification.

Control / Requirement	Traceability & Evidence Reference	Status
AC-2 Account Management Create, enable, modify, disable, and remove information system accounts.	Issues (16): - Cross Site Scripting (Reflected) - Path Traversal - Absence of Anti-CSRF Tokens - File Sensibile Esposto (.idea/workspace.xml)	TECHNICAL GAP
AC-6 Least Privilege Employ the principle of least privilege, allowing only authorized accesses necessary for organizational missions.	Issues (15): - Cross Site Scripting (Reflected) - Path Traversal - Absence of Anti-CSRF Tokens	TECHNICAL GAP
AC-12 Session Termination Automatically terminate a user session after defined conditions.	Issues (16): - Cross Site Scripting (Reflected) - Absence of Anti-CSRF Tokens - GDPR Cookie Consent Missing - Security Headers Analysis - Grade F	TECHNICAL GAP
AU-2 Event Logging Determine that the information system is capable of auditing the needed events.	Issues (13): - Cross Site Scripting (Reflected)	TECHNICAL GAP
CM-6 Configuration Settings Configure the security settings of products to the most restrictive mode consistent with operational requirements.	Issues (2): - Header Sicurezza Mancanti - [OpenDB Match] Nginx Misconfiguration: Server: nginx/1.19.0	REVIEW
CM-3 Configuration Change Control Document, approve, and track changes to the information system.	No direct technical deviations identified.	NOT DETECTED
SI-2 Flaw Remediation Identify, report, and correct information system flaws; install security-relevant software and firmware updates.	Issues (3): - [OpenDB Match] Nginx Misconfiguration: Server: nginx/1.19.0 - [OpenDB Match] PHP 7.x EOL Critical Risks: Framework: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1 - PHP 5.6.40 Obsoleto	TECHNICAL GAP
SI-10 Information Input Validation Check the validity of information inputs.	Issues (21): - Cross Site Scripting (Reflected) - Absence of Anti-CSRF Tokens - SQL Injection - MySQL - Path Traversal [...]	TECHNICAL GAP
SI-11 Error Handling Generate error messages that provide information necessary for corrective actions without revealing exploitable details.	Issues (1): - No HTTPS/SSL Error	TECHNICAL GAP
SC-8 Transmission Confidentiality Protect the confidentiality of transmitted information.	Issues (1): - No HTTPS/SSL Error	TECHNICAL GAP
SC-7 Boundary Protection Monitor and control communications at the external managed interfaces and at key internal boundaries.	Issues (14): - Cross Site Scripting (Reflected) - Absence of Anti-CSRF Tokens	TECHNICAL GAP

Control / Requirement	Traceability & Evidence Reference	Status
IA-5 Authenticator Management Manage information system authenticators.	Issues (1): - Absence of Anti-CSRF Tokens	REVIEW
RA-5 Vulnerability Monitoring Monitor and scan for vulnerabilities in the information system and hosted applications.	Issues (19): - [OpenDB Match] PHP 7.x EOL Critical Risks: Framework: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1 - [OpenDB Match] Nginx Misconfiguration: Server: nginx/1.19.0 - Cross Site Scripting (Reflected) - Absence of Anti-CSRF Tokens [...]	TECHNICAL GAP
IR-4 Incident Handling An incident handling capability for security incidents that includes preparation, detection, analysis, containment, eradication, and recovery.	Issues (3): - Missing Anti-clickjacking Header - Absence of Anti-CSRF Tokens - Security Headers Analysis - Grade F	PARTIAL/GAP

Technical Maturity Assessment: NIST SP 800-53

Maturity Score	1 / 5	Maturity Level	Initial	Coverage	29%
Controls Passed	1	Partial / Review	3	Technical Gaps	10

FIPS 199 Impact Categorization

Security Objective	Potential Impact	Basis
Confidentiality	HIGH	Based on 1 Critical + 21 High findings
Integrity	HIGH	Based on 1 Critical + 21 High findings
Availability	HIGH	Based on 1 Critical + 21 High findings

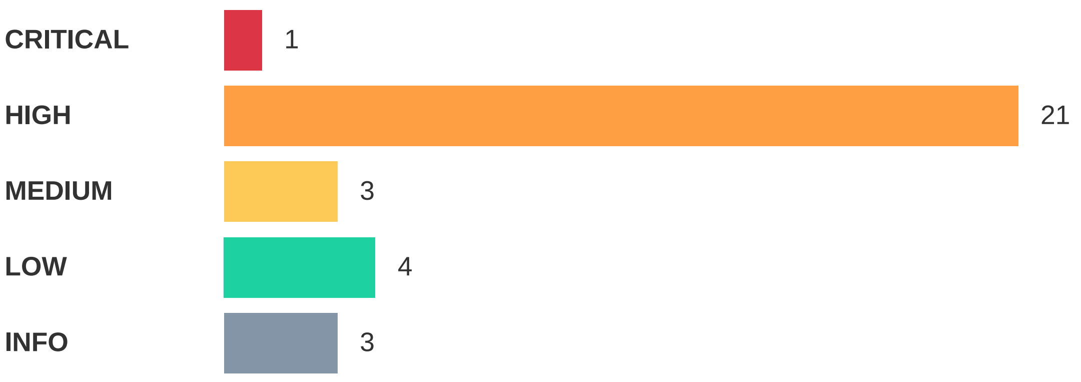
Plan of Action & Milestones (POA&M) Template

The following POA&M format is aligned with NIST SP 800-53 CA-5 and OMB guidance. Populate with remediation details and submit to the Authorizing Official (AO).

ID	Weakness	Control(s)	POC	Resources	Completion	Milestone	Status
1	SQL Injection - MySQL	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open
2	SQL Injection - MySQL	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open
3	SQL Injection - MySQL	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open
4	SQL Injection - MySQL	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open
5	Cross Site Scripting (Reflec	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open
6	Cross Site Scripting (Reflec	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open
7	Cross Site Scripting (Reflec	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open
8	Cross Site Scripting (Reflec	[Map]	[Assign]	[Est.]	[Date]	[Date]	Open

4.G RISK DISTRIBUTION GRAPH

Risk Distribution Graph



5. METHODOLOGY, TEST TYPES & ATTACK COVERAGE

Assessment Timeline & Toolchain

Observed telemetry: 258 HTTP requests, 16 mapped points, 32 subdomains, and 32 findings.

1. Asset Discovery

Subdomains, directories, and JavaScript asset analysis.
- Subfinder [Executed]: Fast passive subdomain enumeration.
- Directory Fuzzing (FFUF) [Configured]: High-performance directory/file brute-forcing.
- Deep JS Analysis [Executed]: JavaScript inspection for exposed endpoints, secrets, and client-side attack surface.
- Recursive Subdomain Scan [Executed]: Discovered subdomains are included in deeper vulnerability analysis.

2. Service & Fingerprint Analysis

Service exposure mapping and vulnerable component intelligence.
- Service Enumeration [Configured]: Open service and version discovery for externally reachable hosts.
- Technology Fingerprinting [Executed]: Software/version inference with vulnerable component correlation.
- Exploit Feasibility Review [Executed]: Evidence-based validation of likely exploit paths and impact.

3. Crawling & Attack Surface Mapping

State-aware and legacy crawling for endpoint coverage.
- Surgical State-Graph Crawler [Executed]: Maps forms, flows, and interactive states for dynamic applications.
- Deep JS Scanner (SPA) [Executed]: Headless execution for DOM attack vectors and hidden endpoints.
- Classic Legacy Spider [Executed]: Traditional href crawling used as compatibility fallback.

4. DAST & Active Verification

Automated dynamic analysis for web-layer security controls.
- OWASP ZAP (Daemon) [Executed]: Advanced DAST integration (v2.17.0). Daemon settings, API key, and port orchestration are managed by Scan Manager.
- Nuclei Engine [Available]: Template-driven detection of known exposures and misconfigurations.

5. Active Injection Modules

Targeted exploit simulation and payload validation.
- SQLMap [Executed]: SQL Injection detection and verification.
- XSStrike [Executed]: Context-aware XSS fuzzing and payload validation.
- Commix [Available]: Command Injection detection for server-side execution vectors.

6. Risk Scoring & Reporting

Consolidation of findings, risk rating, and remediation roadmap.
- Passive Compliance Analysis [Executed]: GDPR/NIST-oriented passive checks and header posture analysis.
- Executive Risk Conclusion [Completed]: Executive risk statement with technical evidence and priority actions.

Assessment Methodology

The evaluation process follows recognized VA/PT practices aligned to NIST SP 800-115, OSSTMM and OWASP guidance. Activities include reconnaissance, fingerprinting, misconfiguration review, vulnerability validation and remediation guidance.

- Black-Box: external perspective without privileged internals.
- Grey-Box: targeted checks with limited context when scope data is provided.
- White-Box: code/configuration review methodology available for explicitly authorized engagements.
- All intrusive checks are executed under controlled conditions and written authorization.

Attack Vectors Executed

- SQL Injection
- SQL Injection (Boolean)
- SQL Injection (Blind)
- SQL Injection (Out of Band)
- Cross-Site Scripting (Reflected/Stored)
- Cross-Site Scripting (Blind)
- Command Injection
- Command Injection (Blind)
- Local File Inclusion
- Remote File Inclusion
- Remote File Inclusion (Out of Band)
- Code Evaluation
- Code Evaluation (Out of Band)
- Server-Side Template Injection
- HTTP Header Injection
- Open Redirection
- Expression Language Injection
- XML External Entity
- XML External Entity (Out of Band)
- Server-Side Request Forgery (Pattern Based)
- Server-Side Request Forgery (DNS)
- File Upload Security Validation
- Reflected File Download
- Insecure Reflected Content
- Web App Fingerprinting
- HTTP Methods Misconfiguration
- Cross-Origin Resource Sharing (CORS) Misconfiguration
- WebDAV Exposure
- Windows Short Filename Enumeration
- RoR Code Execution Checks

Detected in this assessment

- Absence of Anti-CSRF Tokens
- Config
- Content Security Policy (CSP) Header Not Set
- Critical
- Cross Site Scripting (Reflected)
- Email Security
- GDPR Contact Missing
- GDPR Cookie Consent Missing
- Missing Anti-clickjacking Header
- Path Traversal
- SQL Injection - MySQL
- Security

5.C METHODOLOGY REFERENCES

References Methodologies and Techniques Used

NIST SP 800-115

`https://csrc.nist.gov/pubs/sp/800/115/final`

OSSTMM 3

`https://www.isecom.org/OSSTMM.3.pdf`

OWASP Web Security Testing Guide (WSTG)

`https://owasp.org/www-project-web-security-testing-guide/`

OWASP Testing Guide v4

`https://owasp.org/www-pdf-archive/OWASP_Testing_Guide_v4.pdf`

PTES

`http://www.pentest-standard.org/index.php/Main_Page`

OWASP Top 10

`https://owasp.org/www-project-top-ten/`

5.D EVIDENCE REGISTER

Evidence hashes are computed from finding metadata and captured evidence to support integrity and traceability.

ID	Title	Severity	Location	Evidence Hash
DA7DA3A14E8D	SQL Injection - MySQL	High	http://testphp.vulnweb.com/userinfo.php	da7da3a14e8d668504adb6afe9c6bde8
670A0E0DC8EC	SQL Injection - MySQL	High	http://testphp.vulnweb.com/secured/newus	670a0e0dc8ecfc7c2475087dde986cb1
F7D9D24FEEE1	SQL Injection - MySQL	High	http://testphp.vulnweb.com/search.php?te	f7d9d24feee1b1ff1005583b63769e18
6DEFBDB25D52	SQL Injection - MySQL	High	http://testphp.vulnweb.com/search.php?te	6defbdb25d52583d1bfa1ad10707e562
8EA68CC5D550	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/showimage.php	8ea68cc5d550ccccbfcc99c0e4691804
9D76B8AC6E59	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/product.php?p	9d76b8ac6e5984a4574a1bcdde51eede
B0328D24BC33	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/listproducts.	b0328d24bc337665a38003ef48844b90
DB00E0DCF94A	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/listproducts.	db00e0dcf94a5d1671b89c35cb53bfb1
D092DF8C647C	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/hpp/params.ph	d092df8c647c364aa70f80b9c8bd758b
801A433E5994	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/hpp/params.ph	801a433e5994eb0a5732e9596b69f32e
06B303A2AAA5	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/artists.php?a	06b303a2aaa57e8113a2ab53a6de37dc
4DFC3E55ABE9	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/userinfo.php	4dfc3e55abe9553b874717e67514cd45
317DE48D856B	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/secured/newus	317de48d856bd75f3d5d15d391bdfa0d
E60311E9C8D0	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/guestbook.php	e60311e9c8d023d6c1fc13b17dab0767
F0660C00F92D	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/search.php?te	f0660c00f92dae504eec71d4592b6738
188615A7B607	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/search.php?te	188615a7b60752bfe05f3ac488cd0c7b
479DB7EB4D95	Cross Site Scripting (Reflected)	High	http://testphp.vulnweb.com/hpp/?pp=%22	479db7eb4d9593fec2a796f9b010b953
60839145EEC7	Path Traversal	High	http://testphp.vulnweb.com/cart.php	60839145eec7e06b667ac6c3d763e7ce
367613FBD6C8	Missing Anti-clickjacking Header	Info	http://testphp.vulnweb.com/disclaimer.ph	367613fbd6c8dd4f5287d07302fcc30e

5.D EVIDENCE REGISTER (CONTINUED)

ID	Title	Severity	Location	Evidence Hash
481422560A5A	Content Security Policy (CSP) Header Not Set	Info	http://testphp.vulnweb.com/high	481422560a5a04eb97405865293c95c2
F65E59B6F7C3	Absence of Anti-CSRF Tokens	Info	http://testphp.vulnweb.com/	f65e59b6f7c31caf13b163e985165562
4A05DC85855C	File Sensibile Esposto (.idea/workspace.xml)	High	http://testphp.vulnweb.com/	4a05dc85855c45cf834ccaf6435ee56a
14843F7D3CA9	Security Headers Analysis - Grade F	Medium	http://testphp.vulnweb.com/	14843f7d3ca9c90e14de6a90fa9c6f71
	GDPR Cookie Consent Missing	Medium	http://testphp.vulnweb.com/	
33C442D1404F	[GDPR Art. 37-39] Contatto Privacy/DPO Assent	Medium	http://testphp.vulnweb.com/	33c442d1404f38393b3202f13b0bb527
8DC46220E9AB	[OpenDB Match] PHP 7.x EOL Critical Risks: Fr	High	http://testphp.vulnweb.com/	8dc46220e9abc24d8ac127a7d4fb8a27
8938AD8A573A	Header Sicurezza Mancanti	Low	http://testphp.vulnweb.com/	8938ad8a573aa9f6a72b939e14b5a951
5633E8B5B051	PHP 5.6.40 Obsoleto	Critical	http://testphp.vulnweb.com/	5633e8b5b0517dc3436e8a11a420ee9d
7408AA3404C2	[OpenDB Match] Nginx Misconfiguration: Server	Low	http://testphp.vulnweb.com/	7408aa3404c2e0818b8d22ad284912aa
AD9FAE91BA30	No HTTPS/SSL Error	High	http://testphp.vulnweb.com/	ad9fae91ba307f30dd86276cd0804c17
FE57864D8A76	Record SPF Mancante	Low	http://testphp.vulnweb.com/	fe57864d8a76c5715fa8c55671a43747
E6E7F43EF5E3	Record DMARC Mancante	Low	http://testphp.vulnweb.com/	e6e7f43ef5e3370ddf76231a64983bbf

6. DETAILED TECHNICAL FINDINGS

1. PHP 5.6.40 Obsoleto CRITICAL

Description: PHP legacy estremamente vulnerabile.

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `5633E8B5B051`

First Observed 2026-02-08 00:49:51

Method: GET

Impact: Critical System Compromise: Full RCE or Database Access.

Risk Score: 9.5

CVSS: Risk score inferred from severity: Critical (9.5)

Evidence Hash: `5633e8b5b0517dc3436e8a11a420ee9d59d0b556a00df61ce911b794aaf07ec6`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

2. SQL Injection - MySQL HIGH

Description: SQL injection may be possible.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `DA7DA3A14E8D`

First Observed 2026-02-08 01:03:48

Tool: OWASP ZAP

Method: GET

Parameter: uname

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `da7da3a14e8d668504adb6afe9c6bde8e36efa6ed7e7d7bb6f994c25a9fd8be9`

Location:	`http://testphp.vulnweb.com/userinfo.php`
Occurrences:	2 total instances

- http://testphp.vulnweb.com/userinfo.php

Proof of Concept / Technical Evidence:

```
You have an error in your SQL syntax
```

Recommendation:

Do not trust client side input, even if there is client side validation in place.
In general, type check all data on the server side.
If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'
If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.
If database Stored Procedures can be used, use them.
Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!
Do not create dynamic SQL queries using simple string concatenation.
Escape all data received from the client.
Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input.
Apply the principle of least privilege by using the least privileged database user possible.
In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.
Grant the minimum database access that is necessary for the application.

3. SQL Injection - MySQL HIGH

Description:	SQL injection may be possible.
Validation:	Medium. Evidence gathered through controlled testing workflow.
Finding ID:	`670A0E0DC8EC`
First Observed	2026-02-08 01:03:41
Tool:	OWASP ZAP
Method:	GET
Parameter:	uuname
Impact:	Severe Business Risk: Sensitive Data Leak or Admin Takeover.
Confidence:	Medium
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Evidence Hash:	`670a0e0dc8ecfc7c2475087dde986cb14bbe05640e8c724b3bbed6973bc1c318`
Location:	`http://testphp.vulnweb.com/secured/newuser.php`
Occurrences:	2 total instances

- http://testphp.vulnweb.com/secured/newuser.php

Proof of Concept / Technical Evidence:

```
You have an error in your SQL syntax
```

Recommendation:

Do not trust client side input, even if there is client side validation in place.
In general, type check all data on the server side.
If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'
If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.
If database Stored Procedures can be used, use them.
Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!
Do not create dynamic SQL queries using simple string concatenation.
Escape all data received from the client.
Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input.
Apply the principle of least privilege by using the least privileged database user possible.
In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.
Grant the minimum database access that is necessary for the application.

4. SQL Injection - MySQL

HIGH

Description: SQL injection may be possible.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `F7D9D24FEEE1`

First Observed2026-02-08 01:03:34

Tool: OWASP ZAP

Method: GET

Parameter: searchFor

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash`f7d9d24feee1b1ff1005583b63769e18f470cf30d1be032f3cb577095a32e895`

Location: `http://testphp.vulnweb.com/search.php?test=query`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/search.php?test=query

Proof of Concept / Technical Evidence:

```
You have an error in your SQL syntax
```

Recommendation:

Do not trust client side input, even if there is client side validation in place.
In general, type check all data on the server side.
If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'
If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.
If database Stored Procedures can be used, use them.
Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent

functionality!
Do not create dynamic SQL queries using simple string concatenation.
Escape all data received from the client.
Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input.
Apply the principle of least privilege by using the least privileged database user possible.
In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.
Grant the minimum database access that is necessary for the application.

5. SQL Injection - MySQL HIGH

Description:	SQL injection may be possible.
Validation:	Medium. Evidence gathered through controlled testing workflow.
Finding ID:	`6DEFBDB25D52`
First Observed	2026-02-08 01:03:32
Tool:	OWASP ZAP
Method:	GET
Parameter:	test
Impact:	Severe Business Risk: Sensitive Data Leak or Admin Takeover.
Confidence:	Medium
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Evidence Hash:	`6defbdb25d52583d1bfa1ad10707e56246834b0e5f0c7c6e1975c24c1fe92f86`
Location:	`http://testphp.vulnweb.com/search.php?test=%27`
Occurrences:	2 total instances

- http://testphp.vulnweb.com/search.php?test=%27

Proof of Concept / Technical Evidence:

```
You have an error in your SQL syntax
```

Recommendation:

Do not trust client side input, even if there is client side validation in place.
In general, type check all data on the server side.
If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'
If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.
If database Stored Procedures can be used, use them.
Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!
Do not create dynamic SQL queries using simple string concatenation.
Escape all data received from the client.
Apply an 'allow list' of allowed characters, or a 'deny list' of disallowed characters in user input.
Apply the principle of least privilege by using the least privileged database user possible.
In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.

Grant the minimum database access that is necessary for the application.

6. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Low. Evidence gathered through controlled testing workflow.

Finding ID: `8EA68CC5D550`

First Observed2026-02-08 01:02:12

Tool: OWASP ZAP

Method: GET

Parameter: file

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Low

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `8ea68cc5d550ccccbfcc99c0e46918048bcb420d4624c5ede0da7292b41f0e77`

Location: `http://testphp.vulnweb.com/showimage.php?file=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/showimage.php?file=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the

HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

7. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `9D76B8AC6E59`

First Observed: 2026-02-08 01:02:09

Tool: OWASP ZAP

Method: GET

Parameter: pic

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `9d76b8ac6e5984a4574a1bcdde51eedea202596734dbdcd941a410db565bb640`

Location: `http://testphp.vulnweb.com/product.php?pic=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/product.php?pic=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design

Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.

For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.

Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design

For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

8. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.
When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.
Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.
Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `B0328D24BC33`

First Observed: 2026-02-08 01:02:07

Tool: OWASP ZAP

Method: GET

Parameter: cat

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `b0328d24bc337665a38003ef48844b90c7c4ef8b3cae91844507361e40fd4d6c`

Location: `http://testphp.vulnweb.com/listproducts.php?cat=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/listproducts.php?cat=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple

encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

9. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation:	Medium. Evidence gathered through controlled testing workflow.
Finding ID:	`DB00E0DCF94A`
First Observed	2026-02-08 01:02:02
Tool:	OWASP ZAP
Method:	GET
Parameter:	artist
Impact:	Severe Business Risk: Sensitive Data Leak or Admin Takeover.
Confidence:	Medium
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)

Evidence Hash: `db00e0dcf94a5d1671b89c35cb53bfb1afaeb1ecf38f9e699520df430216f9bc`

Location: `http://testphp.vulnweb.com/listproducts.php?artist=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/listproducts.php?artist=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

10. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `D092DF8C647C`

First Observed 2026-02-08 01:01:59

Tool: OWASP ZAP

Method: GET

Parameter: pp

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `d092df8c647c364aa70f80b9c8bd758b2a24cac740d7ff4a9ac436e79ea35d26`

Location: `http://testphp.vulnweb.com/hpp/params.php?p=valid&pp=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/hpp/params.php?p=valid&pp=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation

For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

11. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.
When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the

vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.
Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `801A433E5994`

First Observed: 2026-02-08 01:01:56

Tool: OWASP ZAP

Method: GET

Parameter: p

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `801a433e5994eb0a5732e9596b69f32e5aace2e83e1b54288d7c1f03f4442418`

Location: `http://testphp.vulnweb.com/hpp/params.php?p=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E&pp=12`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/hpp/params.php?p=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E&pp=12

Proof of Concept / Technical Evidence:

```
<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.

Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

12. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation:	Medium. Evidence gathered through controlled testing workflow.
Finding ID:	`06B303A2AAA5`
First Observed	2026-02-08 01:01:48
Tool:	OWASP ZAP
Method:	GET
Parameter:	artist
Impact:	Severe Business Risk: Sensitive Data Leak or Admin Takeover.
Confidence:	Medium
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)

Evidence Hash: `06b303a2aaa57e8113a2ab53a6de37dc29e498f7c7b23ee8ff83cc98fb1adb68`

Location: `http://testphp.vulnweb.com/artists.php?artist=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/artists.php?artist=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

13. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `4DFC3E55ABE9`

First Observed: 2026-02-08 01:01:29

Tool: OWASP ZAP

Method: GET

Parameter: uname

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `4dfc3e55abe9553b874717e67514cd45130ab8ddda01295d94de25b32c56faa3`

Location: `http://testphp.vulnweb.com/userinfo.php`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/userinfo.php

Proof of Concept / Technical Evidence:

```
'"<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an

encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

14. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.
When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the

vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.
Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `317DE48D856B`

First Observed 2026-02-08 01:01:24

Tool: OWASP ZAP

Method: GET

Parameter: uuname

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash `317de48d856bd75f3d5d15d391bdfa0dfbd2a239bb076ff56e6cf9a9e0cdea13`

Location: `http://testphp.vulnweb.com/secured/newuser.php`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/secured/newuser.php

Proof of Concept / Technical Evidence:

```
</li><scrIpt>alert(1);</scRipt><li>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

15. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.
When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have

his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.
Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.
Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `E60311E9C8D0`

First Observed 2026-02-08 01:01:21

Tool: OWASP ZAP

Method: GET

Parameter: name

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash `e60311e9c8d023d6c1fc13b17dab07677fab92122021c2844657bb9b016fe446`

Location: `http://testphp.vulnweb.com/guestbook.php`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/guestbook.php

Proof of Concept / Technical Evidence:

```
</strong><scrIpt>alert(1);</scRipt><strong>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

16. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `F0660C00F92D`

First Observed2026-02-08 01:01:18

Tool: OWASP ZAP

Method: GET

Parameter: searchFor

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hashf0660c00f92dae504eec71d4592b673836de5792e6f16315d690314ce3effed3

Location: `http://testphp.vulnweb.com/search.php?test=query`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/search.php?test=query

Proof of Concept / Technical Evidence:

```
</h2><scrIpt>alert(1);</scRipt><h2>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

17. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `188615A7B607`

First Observed: 2026-02-08 01:01:13

Tool: OWASP ZAP

Method: GET

Parameter: test

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `188615a7b60752bfe05f3ac488cd0c7ba022bcc03dd315bda26a7faacaea79bc`

Location: `http://testphp.vulnweb.com/search.php?test=%27%22%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/search.php?test=%27%22%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
'"<scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an

encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

18. Cross Site Scripting (Reflected) HIGH

Description: Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.
When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the

vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.
Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: `479DB7EB4D95`

First Observed2026-02-08 01:01:02

Tool: OWASP ZAP

Method: GET

Parameter: pp

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Confidence: Medium

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash:`479db7eb4d9593fec2a796f9b010b953612c0251ae4fc7e7a10d8cc9dc3a0f35`

Location: `http://testphp.vulnweb.com/hpp/?pp=%22%3E%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/hpp/?pp=%22%3E%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

Proof of Concept / Technical Evidence:

```
"><scrIpt>alert(1);</scRipt>
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Phases: Implementation; Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed.

Phase: Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

Phase: Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

19. Path Traversal HIGH

Description: The Path Traversal attack technique allows an attacker access to files, directories, and commands that potentially reside outside the web document root directory. An attacker may manipulate a URL in such a way that the web site will execute or reveal the contents of arbitrary files anywhere on the web server. Any device that exposes an HTTP-based interface is potentially vulnerable to Path Traversal.

Most web sites restrict user access to a specific portion of the file-system, typically called the "web document root" or "CGI root" directory. These directories contain the files intended for user access and the executable necessary to drive web application functionality. To access files or execute commands anywhere on the file-system, Path Traversal attacks will utilize the ability of special-characters sequences.

The most basic Path Traversal attack uses the "../" special-character sequence to alter the resource location requested in the URL. Although most popular web servers will prevent this technique from escaping the web document root, alternate encodings of the "../" sequence may help bypass the security filters. These method variations include valid and invalid Unicode-encoding ("..%u2216" or "..%c0%af") of the forward slash character, backslash characters ("..\") on Windows-based servers, URL encoded characters "%2e%2e%2f"), and double URL encoding ("..%255c") of the backslash character.

Even if the web server properly restricts Path Traversal attempts in the URL path, a web application itself may still be vulnerable due to improper handling of user-supplied input. This is a common problem of web applications that use template mechanisms or load static text from files. In variations of the attack, the original URL parameter value is substituted with the file name of one of the web application's dynamic scripts. Consequently, the results can reveal source code because the file is interpreted as text instead of an executable script. These techniques often employ additional special characters such as the dot (".") to reveal the listing of the current working directory, or "%00" NULL characters in order to bypass rudimentary file extension checks.

Validation:	Low. Evidence gathered through controlled testing workflow.
Finding ID:	`60839145EEC7`
First Observed	2026-02-08 00:51:01
Tool:	OWASP ZAP
Method:	GET
Parameter:	price
Impact:	Severe Business Risk: Sensitive Data Leak or Admin Takeover.
Confidence:	Low
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Evidence Hash	`60839145eec7e06b667ac6c3d763e7ce7a21f66fd1dbcc08dc72f8f198d363eb`
Location:	`http://testphp.vulnweb.com/cart.php`
Occurrences:	2 total instances

- http://testphp.vulnweb.com/cart.php

Proof of Concept / Technical Evidence:

```
Detected during controlled assessment and verification workflow.
```

Recommendation:

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

For filenames, use stringent allow lists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses, and exclude directory separators such as "/". Use an allow list of allowable file extensions.

Warning: if you attempt to cleanse your data, then do so that the end result is not in the form that can be dangerous. A sanitizing mechanism can remove characters such as '.' and ';' which may be required for some exploits. An attacker can try to fool the sanitizing mechanism into "cleaning" data into a dangerous form. Suppose the attacker injects a '.' inside a filename (e.g. "sensi.tiveFile") and the sanitizing mechanism removes the character resulting in the valid filename, "sensitiveFile". If the input data are now assumed to be safe, then the file may be compromised.

Inputs should be decoded and canonicalized to the application's current internal representation before being validated. Make sure that your application does not decode the same input twice. Such errors could be used to bypass allow list schemes by introducing dangerous inputs after they have been checked.

Use a built-in path canonicalization function (such as realpath() in C) that produces the canonical version of the pathname, which effectively removes ".." sequences and symbolic links.

Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.

OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

20. File Sensibile Esposto (.idea/workspace.xml)

HIGH

Description: Accessibile a: http://testphp.vulnweb.com/.idea/workspace.xml

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `4A05DC85855C`

First Observed 2026-02-08 00:49:57

Method: GET

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `4a05dc85855c45cf834ccaf6435ee56ab7c0ec1e816edec16b8be19853933753`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

21. [OpenDB Match] PHP 7.x EOL Critical Risks: Framework: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1

HIGH

Description: Status: Rilevamento confermato (Offline DB)

Descrizione: PHP 7.4 è End-of-Life. Esposto a RCE (CVE-2022-31629) e Memory Corruption.

CVE: CVSS 9.8

Fonte: OpenDB Exploit Database (Cached)

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `8DC46220E9AB`

First Observed 2026-02-08 00:49:54

Method: GET

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `8dc46220e9abc24d8ac127a7d4fb8a274bc9e9713f8ea7889350cb52f2cfba89`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

22. No HTTPS/SSL Error

HIGH

Description: Connessione non sicura

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `AD9FAE91BA30`

First Observed2026-02-08 00:49:48

Method: GET

Impact: Severe Business Risk: Sensitive Data Leak or Admin Takeover.

Risk Score: 8.0

CVSS: Risk score inferred from severity: High (8.0)

Evidence Hash: `ad9fae91ba307f30dd86276cd0804c17ac52c692837fe470aad265d44d377934`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

23. Security Headers Analysis - Grade F

MEDIUM

Description: ? HTTP Strict Transport Security (HSTS): Helps protect websites against protocol downgrade attacks and cookie hijacking
? Content Security Policy (CSP): Helps prevent Cross-Site Scripting (XSS) and data injection attacks
? X-Frame-Options: Protects against clickjacking attacks by preventing your site from being embedded in iframes
? X-Content-Type-Options: Prevents browsers from MIME-sniffing a response from the declared content-type
? Referrer Policy: Controls how much referrer information is included with requests
? Permissions Policy: Controls which browser features and APIs can be used in the browser

Validation: Missing 6 security headers. Grade: F (Fail). Evidence gathered through controlled testing workflow.

Finding ID: `14843F7D3CA9`

First Observed2026-02-08 00:49:57

Tool: Header Analyzer

Method: GET

Impact: Moderate Risk: User Session Manipulation or Partial Disclosure.

Confidence: Certain

Risk Score: 5.5

CVSS: Risk score inferred from severity: Medium (5.5)

Evidence Hash: `14843f7d3ca9c90e14de6a90fa9c6f71a54ef1b3368266ab765422c28e1443b6`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Strict-Transport-Security: max-age=31536000; includeSubDomains
Content-Security-Policy: default-src 'self'
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
Referrer-Policy: no-referrer-when-downgrade
Permissions-Policy: camera=(), microphone=(), geolocation=()

24. GDPR Cookie Consent Missing MEDIUM

Description: No valid cookie consent banner was detected on the assessed target. Checks performed: Cookie Policy link, script vendors (30+ markers), consent DOM elements, and accept/reject controls.

Validation: Nessun Cookie Banner, CMP (Consent Management Platform) o meccanismo di consenso rilevato. Evidence gathered through controlled testing workflow.

Tool: GDPR Compliance Scanner

Method: GET

Impact: Moderate Risk: User Session Manipulation or Partial Disclosure.

Confidence: Firm

Risk Score: 5.5

CVSS: Risk score inferred from severity: Medium (5.5)

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit Checks performed: Cookie Policy link, script vendors (30+ markers), consent DOM elements, and accept/reject controls.
```

Recommendation:

1. Implement a certified CMP (e.g., Cookiebot, OneTrust, iubenda).
2. Block all tracking scripts before consent is granted.
3. Provide granular consent controls by cookie category.
4. Store auditable proof of consent, including timestamp and preference state.

25. [GDPR Art. 37-39] Contatto Privacy/DPO Assente MEDIUM

Description: Non è stato rilevato un contatto esplicito per la privacy (DPO, privacy@, ecc.)

Validation: Nessun indirizzo email privacy@, dpo@ o link a modulo contatto privacy trovato. Evidence gathered through controlled testing workflow.

Finding ID: `33C442D1404F`

First Observed 2026-02-08 00:49:57

Tool: GDPR Compliance Scanner

Method: GET

Impact: Moderate Risk: User Session Manipulation or Partial Disclosure.

Confidence: Firm

Risk Score: 5.5

CVSS: Risk score inferred from severity: Medium (5.5)

Evidence Hash: `33c442d1404f38393b3202f13b0bb52708435be62f1fffa3d6f88d2d13f2eb4a`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

1. Create a dedicated privacy contact email (e.g., privacy@domain.com, dpo@domain.com)
2. Publish the contact details in the Privacy Notice
3. If a DPO is mandatory, appoint and register the DPO with the competent supervisory authority

26. Header Sicurezza Mancanti LOW

Description: Strict-Transport-Security
Content-Security-Policy
X-Frame-Options

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `8938AD8A573A`

First Observed 2026-02-08 00:49:54

Method: GET

Impact: Low Risk: Information Gathering or Best Practice Violation.

Risk Score: 3.1

CVSS: Risk score inferred from severity: Low (3.1)

Evidence Hash: `8938ad8a573aa9f6a72b939e14b5a951480074b434740d0c3e30bfa677a1e78c`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

27. [OpenDB Match] Nginx Misconfiguration: Server: nginx/1.19.0 LOW

Description: Status: Rilevamento confermato (Offline DB)

Descrizione: Verificare settings per buffer overflow e header exposure.

CVE: N/A

Fonte: OpenDB Exploit Database (Cached)

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `7408AA3404C2`

First Observed 2026-02-08 00:49:51

Method: GET

Impact: Low Risk: Information Gathering or Best Practice Violation.

Risk Score: 3.1

CVSS: Risk score inferred from severity: Low (3.1)

Evidence Hash: `7408aa3404c2e0818b8d22ad284912aaf8846007779e692f67854eacd28ed71e`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

28. Record SPF Mancante LOW

Description: Rischio SPAM/Spoofing

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `FE57864D8A76`

First Observed 2026-02-08 00:49:48

Method: GET

Impact: Low Risk: Information Gathering or Best Practice Violation.

Risk Score: 3.1

CVSS: Risk score inferred from severity: Low (3.1)

Evidence Hash: `fe57864d8a76c5715fa8c55671a43747c98f3269bed9d678fee903fe20b19082`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

29. Record DMARC Mancante LOW

Description: Rischio BEC limitato

Validation: Observed. Evidence gathered through controlled testing workflow.

Finding ID: `E6E7F43EF5E3`

First Observed 2026-02-08 00:49:48

Method: GET

Impact: Low Risk: Information Gathering or Best Practice Violation.

Risk Score: 3.1

CVSS: Risk score inferred from severity: Low (3.1)

Evidence Hash: `e6e7f43ef5e3370ddf76231a64983bbf671e59c8c4059f073edf3f90f9f972cb`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

30. Missing Anti-clickjacking Header INFO

Description: The response does not protect against 'ClickJacking' attacks. It should include either Content-Security-Policy with 'frame-ancestors' directive or X-Frame-Options.

Validation: Medium. Evidence gathered through controlled testing workflow.

Finding ID: 367613FBD6C8

First Observed 2026-02-08 00:50:10

Tool: OWASP ZAP

Method: GET

Parameter: header-x-frame

Impact: Low Risk: Information Gathering or Best Practice Violation.

Confidence: Medium

Risk Score: 0.0

CVSS: Risk score inferred from severity: Info (0.0)

Evidence Hash: 367613fbd6c8dd4f5287d07302fcc30ecb4281ad4bd76738c21e5ab3d5da2854

Location: http://testphp.vulnweb.com/disclaimer.php

Occurrences: 2 total instances

- http://testphp.vulnweb.com/disclaimer.php

Proof of Concept / Technical Evidence:

```
Detected during controlled assessment and verification workflow.
```

Recommendation:

Modern Web browsers support the Content-Security-Policy and X-Frame-Options HTTP headers. Ensure one of them is set on all web pages returned by your site/app.
If you expect the page to be framed only by pages on your server (e.g. it's part of a FRAMESET) then you'll want to use SAMEORIGIN, otherwise if you never expect the page to be framed, you should use DENY. Alternatively consider implementing Content Security Policy's "frame-ancestors" directive.

31. Content Security Policy (CSP) Header Not Set INFO

Description: Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware. CSP provides a set of standard HTTP headers that allow website owners to declare approved sources of content that browsers should be allowed to load on that page ? covered types are JavaScript, CSS, HTML frames, fonts, images and embeddable objects such as Java applets, ActiveX, audio and video files.

Validation: High. Evidence gathered through controlled testing workflow.

Finding ID: 481422560A5A

First Observed 2026-02-08 00:50:10

Tool: OWASP ZAP

Method: GET

Parameter: header-csp

Impact: Low Risk: Information Gathering or Best Practice Violation.

Confidence: High

Risk Score: 0.0

CVSS: Risk score inferred from severity: Info (0.0)

Evidence Hash: `481422560a5a04eb97405865293c95c2de99368050ec2850811a7eede42a231e`

Location: `http://testphp.vulnweb.com/high`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/high

Proof of Concept / Technical Evidence:

```
Detected during controlled assessment and verification workflow.
```

Recommendation:

Ensure that your web server, application server, load balancer, etc. is configured to set the Content-Security-Policy header.

32. Absence of Anti-CSRF Tokens INFO

Description: No Anti-CSRF tokens were found in a HTML submission form.

A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf.

CSRF attacks are effective in a number of situations, including:
* The victim has an active session on the target site.
* The victim is authenticated via HTTP auth on the target site.
* The victim is on the same local network as the target site.

CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.

Validation: Low. Evidence gathered through controlled testing workflow.

Finding ID: `F65E59B6F7C3`

First Observed: 2026-02-08 00:50:10

Tool: OWASP ZAP

Method: GET

Parameter: csrf-token

Impact: Low Risk: Information Gathering or Best Practice Violation.

Confidence: Low

Risk Score: 0.0

CVSS: Risk score inferred from severity: Info (0.0)

Evidence Hash: `f65e59b6f7c31caf13b163e9851655624f0b6acc8cbdcf09dee1944574b1d115`

Location: `http://testphp.vulnweb.com/`

Occurrences: 2 total instances

- http://testphp.vulnweb.com/

Proof of Concept / Technical Evidence:

```
<form action="search.php?test=query" method="post">
```

Recommendation:

Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, use anti-CSRF packages such as the OWASP CSRFGuard.

Phase: Implementation
Ensure that your application is free of cross-site scripting issues, because most CSRF defenses can be bypassed using attacker-controlled script.

Phase: Architecture and Design
Generate a unique nonce for each form, place the nonce into the form, and verify the nonce upon receipt of the form. Be sure that the nonce is not predictable (CWE-330).
Note that this can be bypassed using XSS.

Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation.
Note that this can be bypassed using XSS.

Use the ESAPI Session Management control.
This control includes a component for CSRF.

Do not use the GET method for any request that triggers a state change.

Phase: Implementation
Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.

7. REMEDIATION TRACKING PLAN

This section provides a structured remediation plan with assigned ownership, priority, and target SLA based on finding severity. Deadlines follow industry-standard timeframes aligned with PCI DSS and NIST guidelines.

Remediation SLA Reference

Severity	Priority	Target SLA	Guidance
Critical	P1	24-72 hours	Immediate containment. Emergency patch. Executive escalation required.
High	P2	7-14 days	Priority remediation in next change window. Verify within 14 days.
Medium	P3	30-60 days	Scheduled remediation. Include in next sprint/maintenance cycle.
Low	P4	90 days	Address during regular maintenance. Monitor for escalation.
Info	P5	Best effort	Informational. Consider hardening. No immediate action required.

Finding Remediation Register

#	Finding	Sev.	Prio	SLA	Owner	Status	Deadline	Verified
1	PHP 5.6.40 Obsoleto	Critical	P1	24-72 hours	[Assign]	Open	[Set]	[ ]
2	SQL Injection - MySQL	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
3	SQL Injection - MySQL	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
4	SQL Injection - MySQL	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
5	SQL Injection - MySQL	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
6	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
7	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
8	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
9	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
10	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
11	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
12	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
13	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
14	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
15	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
16	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
17	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
18	Cross Site Scripting (Reflected)	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
19	Path Traversal	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
20	File Sensibile Esposto (.idea/wo	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
21	[OpenDB Match] PHP 7.x EOL Criti	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
22	No HTTPS/SSL Error	High	P2	7-14 days	[Assign]	Open	[Set]	[ ]
23	Security Headers Analysis - Grad	Medium	P3	30-60 days	[Assign]	Open	[Set]	[ ]

7. REMEDIATION TRACKING (CONTINUED)

#	Finding	Sev.	Prio	SLA	Owner	Status	Deadline	Verified
24	GDPR Cookie Consent Missing	Medium	P3	30-60 days	[Assign]	Open	[Set]	[ ]
25	[GDPR Art. 37-39] Contatto Priva	Medium	P3	30-60 days	[Assign]	Open	[Set]	[ ]
26	Header Sicurezza Mancanti	Low	P4	90 days	[Assign]	Open	[Set]	[ ]
27	[OpenDB Match] Nginx Misconfigur	Low	P4	90 days	[Assign]	Open	[Set]	[ ]
28	Record SPF Mancante	Low	P4	90 days	[Assign]	Open	[Set]	[ ]
29	Record DMARC Mancante	Low	P4	90 days	[Assign]	Open	[Set]	[ ]
30	Missing Anti-clickjacking Header	Info	P5	Best effort	[Assign]	Open	[Set]	[ ]
31	Content Security Policy (CSP) He	Info	P5	Best effort	[Assign]	Open	[Set]	[ ]
32	Absence of Anti-CSRF Tokens	Info	P5	Best effort	[Assign]	Open	[Set]	[ ]

8. RESIDUAL RISK STATEMENT

This section documents the anticipated residual risk after implementation of all recommended remediation actions. Residual risk is the exposure that remains after controls and mitigations are applied.

Current Risk Posture

Total Findings	32
Critical + High Findings	22
Current Overall Risk	CRITICAL

Expected Residual Risk (Post-Remediation)

Expected risk after full remediation	MEDIUM
Remaining findings (Low/Info only)	7

Residual Risk Factors (Inherent Limitations)

- Zero-day vulnerabilities not detectable by current testing methods.
- Business logic flaws requiring authenticated/contextual testing beyond scope.
- Supply chain risks in third-party components not fully enumerable.
- Social engineering and insider threat vectors (out of technical VA/PT scope).
- Configuration drift between assessment date and remediation completion.
- Evolving threat landscape may introduce new attack vectors post-assessment.
- Cloud/SaaS provider shared-responsibility controls not directly testable.

Recommendation: Schedule a follow-up reassessment within 90 days of completing Critical/High remediations to validate effectiveness. Annual full-scope VA/PT is recommended as part of continuous security posture management per ISO 27001 Clause 10.2 and NIST SP 800-53 CA-2.

A. GLOSSARY OF TERMS & ABBREVIATIONS

Term	Definition
ACSC	Australian Cyber Security Centre
ASVS	Application Security Verification Standard (OWASP)
Black-Box	Testing without prior knowledge of internal systems
CORS	Cross-Origin Resource Sharing
CSRF	Cross-Site Request Forgery
CVE	Common Vulnerabilities and Exposures - publicly disclosed security flaws
CVSS	Common Vulnerability Scoring System (v2.0/v3.1) - standardized severity rating
CWE	Common Weakness Enumeration - software security weakness categorization
DAST	Dynamic Application Security Testing
DPIA	Data Protection Impact Assessment (GDPR Art. 35)
DPO	Data Protection Officer
ePHI	Electronic Protected Health Information (HIPAA)
FedRAMP	Federal Risk and Authorization Management Program
FIPS	Federal Information Processing Standards
GDPR	General Data Protection Regulation (EU 2016/679)
Grey-Box	Testing with limited internal knowledge
HIPAA	Health Insurance Portability and Accountability Act (U.S.)
ISMS	Information Security Management System
ISO 27001	International standard for Information Security Management Systems
LFI/RFI	Local / Remote File Inclusion
MFA	Multi-Factor Authentication
NCSC	National Cyber Security Centre (United Kingdom)
NIST	National Institute of Standards and Technology (U.S. Dept. of Commerce)
OSSTMM	Open Source Security Testing Methodology Manual
OWASP	Open Web Application Security Project
POA&M	Plan of Action and Milestones (NIST/FISMA)
PTES	Penetration Testing Execution Standard
RCE	Remote Code Execution
SAST	Static Application Security Testing
SHA-256	Secure Hash Algorithm 256-bit (evidence integrity)
SLA	Service Level Agreement
SoA	Statement of Applicability (ISO 27001 Annex A)
SOC 2	Service Organization Control 2 - AICPA Trust Services Criteria
SPA	Single Page Application
SQLi	SQL Injection

A. GLOSSARY (CONTINUED)

Term	Definition
SSP	System Security Plan
SSRF	Server-Side Request Forgery
TLP	Traffic Light Protocol - information sharing classification
TLP:RED	Restricted disclosure - authorized recipients only
TSC	Trust Services Criteria (SOC 2)
VA/PT	Vulnerability Assessment and Penetration Test
White-Box	Testing with full source code / configuration access
WSTG	Web Security Testing Guide (OWASP)
XSS	Cross-Site Scripting
XXE	XML External Entity Injection

B. RISK ACCEPTANCE DECLARATION

Following the review of this VA/PT report, the Client organization acknowledges the identified risks and their potential business impact. This section formally documents risk treatment decisions.

Risk Summary

Overall Risk Rating	CRITICAL
Total Findings	32
Critical / High Findings	1 / 21
Medium / Low / Info	3 / 4 / 3

Risk Treatment Decision Register

#	Finding Title	Sev.	Treatment (Mitigate/Accept/Transfer/Avoid)	Business Justification
1	PHP 5.6.40 Obsoleto	Critical	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
2	SQL Injection - MySQL	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
3	SQL Injection - MySQL	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
4	SQL Injection - MySQL	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
5	SQL Injection - MySQL	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
6	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
7	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
8	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
9	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
10	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
11	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
12	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
13	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
14	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
15	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
16	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
17	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
18	Cross Site Scripting (Reflected)	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
19	Path Traversal	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
20	File Sensibile Esposto (.idea/workspac	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
21	[OpenDB Match] PHP 7.x EOL Critical Ri	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
22	No HTTPS/SSL Error	High	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
23	Security Headers Analysis - Grade F	Medium	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	

B. RISK ACCEPTANCE (CONTINUED)

#	Finding Title	Sev.	Treatment	Business Justification
24	GDPR Cookie Consent Missing	Medium	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	
25	[GDPR Art. 37-39] Contatto Privacy/DPO	Medium	[ ]Mit [ ]Acc [ ]Trf [ ]Avd	

Treatment: MITIGATE (implement fix), ACCEPT (retain risk), TRANSFER (insure/outsource), AVOID (discontinue service).
Risk acceptance for Critical/High findings requires executive-level approval and documented business justification per ISO 27001 Clause 6.1.3 and NIST SP 800-37.

Risk Owner Approval

Name: ________________________________ Title: ________________________________

Signature: ____________________________ Date: _______________

ATTESTATION & SIGN-OFF

This Vulnerability Assessment and Penetration Test report has been prepared in accordance with industry-standard methodologies (NIST SP 800-115, OSSTMM 3, OWASP WSTG) and represents the findings observed during the authorized testing window.

The undersigned parties attest that:
1. Testing was conducted within the authorized scope and rules of engagement.
2. All findings have been verified and documented with supporting evidence.
3. Evidence integrity is maintained via SHA-256 hashing of each finding.
4. Compliance mappings are automated technical observations and do not constitute certification.
5. This document is classified TLP:RED; distribution is restricted to named recipients.
6. The assessment represents a point-in-time snapshot and does not guarantee ongoing security.

Lead Auditor / Assessor

Name: ExploitFinder Engine Date: _______________

Signature: ____________________________ Title: ____________________________

QA Reviewer

Name: Cyber Advisory Team Date: _______________

Signature: ____________________________ Title: ____________________________

Client Authorized Representative

Name: ________________________________ Date: _______________

Signature: ____________________________ Title: ____________________________

Client Technical Contact

Name: ________________________________ Date: _______________

Signature: ____________________________ Title: ____________________________

Document ID: ee71f143-f81d-4d97-a022-d2d07c93be0e | Assessment Date: 2026-02-08 00:49:06 | Risk Classification: CRITICAL | Classification: TLP:RED

This attestation confirms review and acceptance of the assessment methodology, findings, and recommendations. Signing does not imply agreement with all findings but acknowledges receipt and review of the complete document.