VA/PT REPORT

VULNERABILITY ASSESSMENT & PENETRATION TEST

PREPARED FOR:

Client Organization / TESTPHP.VULNWEB.COM

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

PREPARED BY:

Cyber Advisory LLC

ExploitFinder Security Team

DOCUMENT ID:	`489a6845-cf22-47c5-bdc5-42a84023cbf4`
DATE:	2026-02-07 17:56:32

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	07/02/2026
QA Reviewer	Cyber Advisory Team	Approved	07/02/2026
Report ID	489a6845-cf22-47c5-bdc5-42a84023cbf4	Version	1.0

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: 489a6845-cf22-47c5-bdc5-42a84023cbf4
- Assessment date: 2026-02-07 17:56:32
- Assets analyzed: 1 IP(s), 32 subdomain(s)
- Total findings: 64 (Critical 2, High 32, Medium 6, Low 18, Info 6)

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	64

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

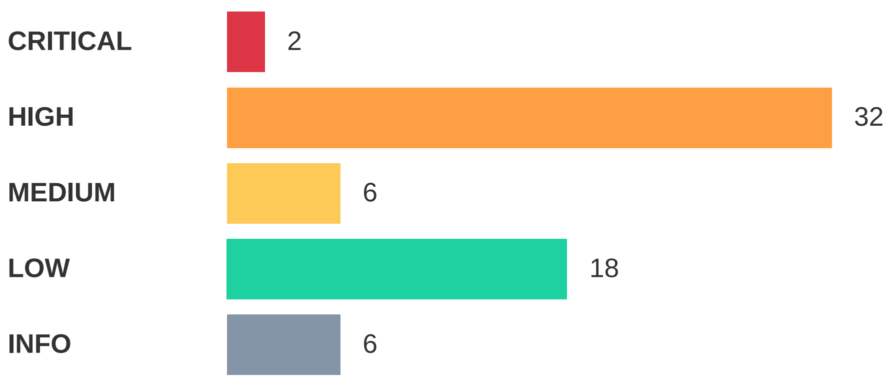
Network Surface Inventory (All Discovered Subdomains)

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

4.N NETWORK SURFACE INVENTORY (CONTINUED)

Host	HTTP Status
host-158.testphp.vulnweb.com	dead
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

Risk Distribution Graph



5. METHODOLOGY, TEST TYPES & ATTACK COVERAGE

Assessment Timeline & Toolchain

Observed telemetry: 691 HTTP requests, 30 mapped points, 32 subdomains, and 64 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
- SQL Injection - MySQL
- Security
- Security Headers Missing

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/`

6. DETAILED TECHNICAL FINDINGS

1. PHP 5.6.40 Obsoleto CRITICAL

Description: PHP legacy estremamente vulnerabile.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 9.5

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

2. PHP 5.6.40 Obsoleto CRITICAL

Description: PHP legacy estremamente vulnerabile.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 9.5

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

3. SQL Injection - MySQL HIGH

Description:	SQL injection may be possible.
Validation:	Medium. Evidence gathered through controlled testing workflow.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=query`

Proof of Concept / 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=query`

Proof of Concept / 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=%27`

Proof of Concept / 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. SQL Injection - MySQL HIGH

Description:	SQL injection may be possible.
Validation:	Medium. Evidence gathered through controlled testing workflow.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=%27`

Proof of Concept / 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.

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:	Low. Evidence gathered through controlled testing workflow.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/showimage.php?file=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E&size=160`

Proof of Concept / 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:	Low. Evidence gathered through controlled testing workflow.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/showimage.php?file=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E&size=160`

Proof of Concept / 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:	Low. Evidence gathered through controlled testing workflow.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/showimage.php?file=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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:	Low. Evidence gathered through controlled testing workflow.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/showimage.php?file=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/product.php?pic=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/product.php?pic=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/listproducts.php?cat=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/listproducts.php?cat=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/listproducts.php?artist=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/listproducts.php?artist=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/artists.php?artist=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

Risk Score: 8.0

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

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

Proof of Concept / 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. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/secured/newuser.php`

Proof of Concept / 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.

20. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/secured/newuser.php`

Proof of Concept / 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.

21. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=query`

Proof of Concept / 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.

22. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=query`

Proof of Concept / 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.

23. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=%27%22%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

24. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/search.php?test=%27%22%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

25. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/guestbook.php`

Proof of Concept / 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.

26. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/guestbook.php`

Proof of Concept / 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.

27. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/hpp/?pp=%22%3E%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

28. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/hpp/?pp=%22%3E%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E`

Proof of Concept / 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.

29. 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.
Risk Score:	8.0
CVSS:	Risk score inferred from severity: High (8.0)
Location:	`http://testphp.vulnweb.com/`

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

30. 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.

Risk Score: 8.0

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:
Verificare la configurazione secondo le best practices di sicurezza.

31. [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.

Risk Score: 8.0

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:
Verificare la configurazione secondo le best practices di sicurezza.

32. [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.

Risk Score: 8.0

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

33. No HTTPS/SSL Error HIGH

Description: Connessione non sicura

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 8.0

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

34. No HTTPS/SSL Error HIGH

Description: Connessione non sicura

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 8.0

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

35. 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.
Risk Score:	5.5
CVSS:	Risk score inferred from severity: Medium (5.5)
Location:	`http://testphp.vulnweb.com/`

Proof of Concept / 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=()

36. 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.
Risk Score:	5.5
CVSS:	Risk score inferred from severity: Medium (5.5)

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

Proof of Concept / 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=()

37. 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.
Risk Score:	5.5
CVSS:	Risk score inferred from severity: Medium (5.5)
Location:	`http://testphp.vulnweb.com/`

Proof of Concept / 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.

38. 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.
Risk Score:	5.5
CVSS:	Risk score inferred from severity: Medium (5.5)
Location:	`http://testphp.vulnweb.com/`

Proof of Concept / 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.

39. [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.

Risk Score: 5.5

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

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

Proof of Concept / 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

40. [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.

Risk Score: 5.5

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

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

Proof of Concept / 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

41. Suspicious Reflected Parameter   LOW

Description: Suspicious reflection on 'searchFor <d3V%09onMoUSeovEr+=+a=prompt,a()>v3dm'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/product.php?pic`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <d3V%09onMoUSeovEr+=+a=prompt,a()>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

42. Suspicious Reflected Parameter   LOW

Description: Suspicious reflection on 'searchFor <d3V%09onMoUSeovEr+=+a=prompt,a()>v3dm'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/product.php?pic`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <d3V%09onMoUSeovEr+=+a=prompt,a()>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

43. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <D3V%09onmOUSeOveR%0a=%0a(prompt)``>v3'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/hpp/?pp`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <D3V%09onmOUSeOveR%0a=%0a(prompt)``>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

44. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <D3V%09onmOUSeOveR%0a=%0a(prompt)``>v3'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/hpp/?pp`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <D3V%09onmOUSeOveR%0a=%0a(prompt)``>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

45. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <htmL%0aoNpOiNtEreNTER+=+confirm()//'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/artists.php?artist`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <htmL%0aoNpOiNtEreNTER+=+confirm()//
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

46. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <htmL%0aoNpOiNtEreNTER+=+confirm()//'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/artists.php?artist`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <htmL%0aoNpOiNtEreNTER+=+confirm()//
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

47. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <A/+/ONPointeRenter%09=%09confirm()>v3'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/listproducts.php?cat`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <A/+/ONPointeRenter%09=%09confirm()>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

48. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <A/+/ONPointeRenter%09=%09confirm()>v3'. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

Location: `http://testphp.vulnweb.com/listproducts.php?cat`

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <A/+/ONPointeRenter%09=%09confirm()>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

49. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <D3V%09onpOiNtEREnter%0d=%0d(confirm)('. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <D3V%09onpOiNtEREnter%0d=%0d(confirm)()>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

50. Suspicious Reflected Parameter LOW

Description: Suspicious reflection on 'searchFor <D3V%09onpOiNtEREnter%0d=%0d(confirm)('. XSS payload did not execute in replay.

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Vulnerable Parameter: searchFor <D3V%09onpOiNtEREnter%0d=%0d(confirm)()>v3dm0s
Payload (Auto-Generated PoC): "><script>alert(1)</script>

[!] VERIFICATION: Payload sent but NOT reflected as-is. Potential false positive or sanitized.
```

Recommendation:
Implement proper input validation and output encoding. Use Content-Security-Policy headers.

51. [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.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:
Verificare la configurazione secondo le best practices di sicurezza.

52. Header Sicurezza Mancanti LOW

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

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:
Verificare la configurazione secondo le best practices di sicurezza.

53. [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.
Risk Score:	3.1
CVSS:	Risk score inferred from severity: Low (3.1)
Location:	`http://testphp.vulnweb.com/`

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

54. Header Sicurezza Mancanti LOW

Description:	Strict-Transport-Security Content-Security-Policy X-Frame-Options
Validation:	Observed. Evidence gathered through controlled testing workflow.
Risk Score:	3.1
CVSS:	Risk score inferred from severity: Low (3.1)
Location:	`http://testphp.vulnweb.com/`

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

55. Record SPF Mancante LOW

Description: Rischio SPAM/Spoofing

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

56. Record SPF Mancante LOW

Description: Rischio SPAM/Spoofing

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

57. Record DMARC Mancante LOW

Description: Rischio BEC limitato

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

58. Record DMARC Mancante LOW

Description: Rischio BEC limitato

Validation: Observed. Evidence gathered through controlled testing workflow.

Risk Score: 3.1

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

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

Proof of Concept / Evidence:

```
Detected during Passive Audit
```

Recommendation:

Verificare la configurazione secondo le best practices di sicurezza.

59. 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.

Risk Score: 0.0

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

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

Proof of Concept / 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.

60. 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.

Risk Score: 0.0

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

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

Proof of Concept / 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.

61. 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.

Risk Score: 0.0

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

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

Proof of Concept / 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.

62. 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.

Risk Score: 0.0

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

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

Proof of Concept / 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.

63. 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.

Risk Score: 0.0

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

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

Proof of Concept / 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.

64. 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.
Risk Score:	0.0
CVSS:	Risk score inferred from severity: Info (0.0)
Location:	`http://testphp.vulnweb.com/`

Proof of Concept / 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.