Web PenTesting Checklist by Joas

https://www.linkedin.com/in/joas-antonio-dos-santos

Wordpress CMS

- Keep WordPress updated: Regularly update the WordPress core, plugins, and themes to protect against known vulnerabilities.
- Test for weak passwords: Ensure strong passwords are used for all user accounts, especially for administrator accounts.
- Check for user enumeration: Test if usernames can be enumerated through the WordPress author archives or other means, and disable user enumeration if possible.
- Test for default admin username: Ensure the default "admin" username is not used, and replace it with a custom username.
- Limit login attempts: Test if login attempts are limited to prevent brute-force attacks, and install a plugin like Login LockDown or Wordfence to enable this functionality if necessary.
- Test for insecure file permissions: Check the permissions of your WordPress files and folders to ensure they are secure and cannot be accessed by unauthorized users.
- Test for XML-RPC vulnerabilities: Test for vulnerabilities related to the XML-RPC feature, such as DDoS or brute-force attacks, and disable it if not needed.
- Test for SQL injection vulnerabilities: Test your WordPress site for SQL injection vulnerabilities by injecting SQL payloads into input fields or URL parameters.
- Test for Cross-Site Scripting (XSS) vulnerabilities: Test your WordPress site for XSS vulnerabilities by injecting JavaScript payloads into input fields or URL parameters.
- Test for Cross-Site Request Forgery (CSRF) vulnerabilities: Test your WordPress site for CSRF vulnerabilities by attempting to perform actions without a valid CSRF token or by using another user's authenticated session.
- Test for vulnerable plugins: Check for known vulnerabilities in your installed plugins using tools like WPScan or by regularly monitoring vulnerability databases.
- Test for vulnerable themes: Check for known vulnerabilities in your installed themes using tools like WPScan or by regularly monitoring vulnerability databases.
- Test for insecure configurations: Check your WordPress configuration (wp-config.php) for insecure settings, such as displaying errors, and secure it by disabling features like error reporting or file editing.
- Check for security best practices: Ensure your site follows WordPress security best practices, such as using HTTPS, disabling directory browsing, or setting secure HTTP headers.
- Use a security plugin: Install a comprehensive security plugin like Wordfence, iThemes Security, or Sucuri to monitor and protect your site from various threats.

Subdomain Takeover

- Enumerate subdomains: Use tools like Sublist3r, Amass, or dnsrecon to discover subdomains associated with your main domain.
- Analyze DNS records: Check DNS records (e.g., CNAME, A, AAAA, MX) for subdomains pointing to external services or expired domains.
- Check HTTP responses: Examine HTTP responses for error messages or status codes that may indicate an unclaimed or expired external service.
- Use online services: Utilize online services such as crt.sh or Censys to gather subdomain and certificate data for your main domain.
- Test common third-party services: Check if subdomains are pointing to common third-party services, such as AWS S3, GitHub Pages, or Heroku, that are susceptible to subdomain takeover attacks.
- Test for dangling CNAME records: Look for dangling CNAME records that point to external services that have been deleted or expired.
- Monitor domain registration: Monitor domain registration information for expired domains that can be taken over.
- Use subdomain takeover tools: Utilize tools like SubOver, Subjack, or tko-subs to automatically identify subdomain takeover vulnerabilities.
- Check for misconfigured DNS settings: Examine DNS settings for misconfigurations that might lead to subdomain takeover vulnerabilities.
- Test for wildcard DNS records: Check for wildcard DNS records that might expose subdomains to takeover attacks.
- Check for abandoned subdomains: Look for abandoned subdomains that still point to unused external services.
- Test for improper redirects: Check if subdomains are improperly redirecting traffic to external services that can be taken over.
- Monitor domain ownership changes: Monitor domain ownership changes for potential takeover opportunities.
- Collaborate with third-party service providers: Work with third-party service providers to ensure proper domain configuration and prevent subdomain takeover.
- Regularly audit subdomain configurations: Periodically review your subdomain configurations to identify and mitigate potential subdomain takeover risks.

IDOR

- Sequential IDs: Analyze sequential numeric IDs or predictable identifiers in URLs, API endpoints, or hidden form fields, and try modifying them to access unauthorized resources.
- User-specific data: Ensure proper authorization checks are in place for user-specific data, such as profiles, orders, or messages, by attempting to access another user's data using your authenticated session.
- Enumerate identifiers: Create multiple accounts with different roles (e.g., admin, user) and compare the object identifiers to identify patterns or correlations.
- Test file uploads: Test file upload functionality and attempt to access uploaded files by guessing or modifying their filenames.
- Test API endpoints: Analyze API endpoints for exposed object references and attempt to access unauthorized resources by modifying request parameters.
- Test hidden form fields: Examine hidden form fields for object references and modify their values to access unauthorized resources.
- Test JSON or XML responses: Analyze JSON or XML responses for exposed object references and attempt to access unauthorized resources by modifying request parameters.
- Test related features: Test related features or modules, such as password reset or email validation, for IDOR vulnerabilities by modifying request parameters.
- Test with different roles: Create accounts with different roles (e.g., admin, user, guest) and attempt to access unauthorized resources using different user sessions.
- Test with unauthenticated sessions: Test if unauthenticated users can access resources by modifying object references in URLs or API endpoints.
- Use web application scanners: Use automated web application scanners, such as Burp Suite or OWASP ZAP, to identify potential IDOR vulnerabilities.
- Analyze access logs: Review server access logs for patterns indicating unauthorized access attempts.
- Manipulate cookies: Manipulate cookies or session tokens to impersonate other users and attempt to access unauthorized resources.
- Test request methods: Test for IDOR vulnerabilities using different HTTP request methods, such as GET, POST, PUT, DELETE, or PATCH.
- Test with URL-encoded or base64-encoded parameters: Try URL-encoded or base64-encoded parameters to bypass input validation or access control checks.

XXE

- Basic external entity: Inject a basic external entity reference to test if the parser resolves it
- External parameter entity: Inject an external parameter entity to bypass input filters.
- Blind XXE (OOB technique): Use Out-of-Band (OOB) techniques to exfiltrate data if the response doesn't display the content of the external entity.
- File inclusion: Attempt to include local or remote files using the SYSTEM identifier to test for arbitrary file inclusion.
- Internal entity expansion: Inject an internal entity with a large number of nested entities to test for a Billion Laughs attack (a type of denial-of-service attack).
- Recursive entity references: Test for recursive entity expansion to identify potential denial-of-service (DoS) vulnerabilities.
- XML bomb: Inject a large XML file with deeply nested elements to test for XML bomb vulnerabilities, which can lead to DoS attacks.
- Error-based XXE: Inject malformed XML with external entity references to trigger errors that reveal sensitive information.
- XML encoding: Try different XML encodings (e.g., UTF-16, UTF-32) to bypass input filters that block specific characters.
- Use CDATA sections: Inject external entity references inside CDATA sections to bypass input filters that remove or escape specific characters.
- Custom entities: Create custom entities with external references to test if the XML parser resolves them.
- Test various content types: Test for XXE vulnerabilities in different content types that support XML, such as SOAP, XHTML, SVG, or RSS.
- Test XML-based file formats: Test for XXE vulnerabilities in XML-based file formats, such as Office Open XML (.docx, .pptx, .xlsx) or OpenDocument (.odt, .ods, .odp).
- Test different HTTP methods: Test for XXE vulnerabilities using different HTTP methods, such as POST, PUT, or PATCH, with XML payloads.
- Test XML-based APIs: Test for XXE vulnerabilities in XML-based APIs, such as XML-RPC or SOAP-based web services.

XSS

- Basic payload injection: Inject simple script tags or HTML tags with JavaScript event handlers into input fields or query parameters. Example: <script>alert(1)</script> or <img src=x onerror=alert(1)>.
- URL encoding: Use URL-encoded payloads to bypass input filters that may block certain characters. Example: %3Cscript%3Ealert(1)%3C%2Fscript%3E.
- Hex encoding: Test with hex-encoded payloads to bypass filters that block specific characters. Example: <scr\x69pt>alert(1)</scr\x69pt>.
- Case variation: Try different letter casing to bypass case-sensitive filters. Example: <ScRiPt>alert(1)</ScRiPt>.
- HTML entity encoding: Inject payloads with HTML entities to evade filters that remove or escape specific characters. Example: &lt;script&gt;alert(1)&lt;/script&gt;.
- Null byte injection: Use null bytes to break out of input restrictions or bypass filters. Example: <scr%00ipt>alert(1)</scr%00ipt>.
- Double encoding: Test with double-encoded payloads to bypass filters that only decode input once. Example: %253Cscript%253Ealert(1)%253C%252Fscript%253E.
- Attribute injection: Attempt to inject payloads within existing HTML tags by closing the current attribute and adding a new one with malicious JavaScript. Example: "><img src=x onerror=alert(1)>.
- JavaScript event handlers: Inject JavaScript event handlers, such as onmouseover, onfocus, or onclick, into various HTML elements to trigger the payload.
- Malformed tags: Test with malformed tags to bypass filters that look for well-formed HTML. Example: <scrip<script>t>alert(1)</scrip</script>t>.
- Using different contexts: Test payloads in various contexts, such as HTML comments, inline JavaScript, or CSS, to bypass context-specific filters.
- Data URI: Inject data URI payloads to bypass certain input filters. Example: <iframe src="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg=="></iframe>.
- SVG payloads: Use Scalable Vector Graphics (SVG) payloads to execute JavaScript in a different context. Example: <svg onload="alert(1)"></svg>.
- Breaking out of JavaScript: Inject payloads that break out of existing JavaScript code and execute malicious scripts.
- Testing error pages: Check if error pages, such as 404 or 500, reflect user input without proper encoding, as these can be used for reflected XSS attacks.

File Upload

- File size limit: Verify that there is an appropriate file size limit in place to prevent large file uploads that could potentially exhaust server resources.
- File type restrictions: Ensure that only allowed file types can be uploaded, and test with disallowed file types to confirm the restrictions are working.
- MIME type validation: Check that the MIME type of uploaded files is being validated and that the system rejects files with incorrect MIME types.
- Filename validation: Test that the system filters and sanitizes filenames to avoid malicious filenames (e.g., "../", ".htaccess") that could lead to security vulnerabilities.
- Malware scanning: Scan uploaded files for malware or viruses using an up-to-date antivirus solution.
- Duplicate file names: Test how the system handles duplicate file names, ensuring that it doesn't overwrite existing files or create security vulnerabilities.
- Upload directory: Verify that the upload directory is secured and not accessible for unauthorized users.
- Permissions: Ensure that proper file and folder permissions are set to prevent unauthorized access, modification, or deletion of uploaded files.
- User authentication: Test if file uploads require proper user authentication and that unauthorized users cannot upload files.
- Image validation: If uploading images, test for potential vulnerabilities related to image processing libraries (e.g., buffer overflows, code injection).
- File content validation: Ensure that the content of the files is validated and doesn't contain malicious code or scripts.
- Maximum file uploads: Test the maximum number of simultaneous file uploads to ensure the system can handle the load without crashing or compromising security.
- Timeouts: Test the system for handling long uploads and confirm that it has appropriate timeouts in place.
- Rate limiting: Verify that the system has rate limiting in place to prevent abuse and denial of service (DoS) attacks.
- Error handling: Test the system's error handling capabilities to ensure that it doesn't leak sensitive information or create security vulnerabilities.
- Cross-site scripting (XSS): Test for potential XSS vulnerabilities related to file uploads, such as the inclusion of malicious scripts within file metadata.
- Path traversal: Test for path traversal vulnerabilities by attempting to upload files with directory traversal characters (e.g., "../") in the file name.
- SQL injection: Test for potential SQL injection vulnerabilities related to file uploads, such as manipulating metadata to include malicious SQL queries.
- Access control: Verify that proper access controls are in place for viewing, editing, or deleting uploaded files.
- Logging and monitoring: Ensure that the system logs and monitors all file upload activities for potential security threats and suspicious behavior.

Cookie Settings

- Insecure transmission: Ensure cookies are sent only over HTTPS connections, to prevent interception by attackers. Set the "Secure" attribute for all cookies.
- Missing HttpOnly attribute: Set the "HttpOnly" attribute to ensure cookies are inaccessible to client-side scripts, reducing the risk of cross-site scripting (XSS) attacks.
- Missing SameSite attribute: Set the "SameSite" attribute to "Strict" or "Lax" to prevent cross-site request forgery (CSRF) attacks by ensuring cookies are only sent with requests originating from the same domain.
- Excessive cookie lifetime: Limit the duration of cookie validity by setting the "Expires" or "Max-Age" attribute. Long-lived cookies pose a greater risk if they are compromised.
- Weak encryption: Use strong encryption algorithms and up-to-date cryptographic libraries to protect sensitive information stored in cookies.
- Insufficiently random session IDs: Ensure session IDs are generated using a strong source of randomness, to prevent session hijacking and guessing attacks.
- Overly permissive cookie domain and path: Limit the scope of cookies by setting the "Domain" and "Path" attributes to specific subdomains or directories, reducing the risk of unauthorized access.
- Storing sensitive information in cookies: Avoid storing sensitive information, such as passwords, API keys, or personally identifiable information (PII) in cookies. Instead, store them server-side and use session IDs to reference the data.
- Unprotected cookie values: Ensure that cookie values are hashed, encrypted, or signed to protect them from being tampered with by attackers.
- Inadequate monitoring and logging: Implement a proper monitoring and logging system to track cookie usage, to help detect and respond to potential security incidents.

SSRF

- Test user-controlled URLs: Identify user-controlled URL inputs and test them with external URLs to see if the server fetches or processes them.
- Test internal IP addresses: Attempt to access internal IP addresses (e.g., 127.0.0.1 or 10.0.0.0/8) or services through user-controlled inputs to check if the server processes them.
- Use URL schemas: Test various URL schemas, such as file://, ftp://, or gopher://, to bypass input validation or access internal resources.
- Test domain resolution: Test if your server resolves domain names to internal IP addresses by using a domain that points to an internal IP address.
- Test URL redirection: Test if the server follows redirects by supplying a URL that redirects to an internal or external resource.
- Test with different HTTP methods: Test SSRF vulnerabilities with various HTTP methods, such as GET, POST, PUT, DELETE, or HEAD.
- Test with malformed URLs: Test with malformed URLs that may bypass input validation, such as using @ to separate credentials or adding extra slashes.
- Test for open ports: Attempt to access open ports on the server or internal network by specifying the target IP and port in the URL.
- Test for Out-of-Band (OOB) data exfiltration: Test if the server can send data to an external domain you control, which may indicate an SSRF vulnerability.
- Test for cloud service metadata: If your site is hosted on a cloud provider, test if the server can access cloud service metadata endpoints, which may expose sensitive information.
- Test with time-based techniques: Use time-based techniques, such as delays or timeouts, to confirm SSRF vulnerabilities when the server response doesn't reveal the fetched content.
- Test for protocol smuggling: Test for protocol smuggling, such as using http:// within an https:// URL, to bypass input validation or access internal resources.
- Test for bypassing URL filtering: Attempt to bypass URL filtering using techniques like URL encoding, double encoding, or mixed case encoding.
- Use web application scanners: Use automated web application scanners, such as Burp Suite or OWASP ZAP, to identify potential SSRF vulnerabilities.
- Test with IPv6 addresses: Test for SSRF vulnerabilities using IPv6 addresses to bypass input validation or access internal resources.

WAF Testing

- Test with OWASP Top Ten attacks: Test for the most common web application vulnerabilities, such as SQLi, XSS, CSRF, and RCE.
- Use WAF testing tools: Utilize tools like Wafw00f, Nmap, or WAPT to identify and test your WAF's capabilities.
- Test for HTTP methods: Test different HTTP methods (GET, POST, PUT, DELETE, etc.) to check if your WAF is properly filtering and blocking malicious requests.
- Test for HTTP protocol violations: Send requests that violate the HTTP protocol to see if your WAF can detect and block them.
- Test with malformed requests: Send malformed requests with invalid or unexpected characters, encoding, or headers to test if your WAF can detect and block them.
- Test for evasion techniques: Test various evasion techniques, such as URL encoding, double encoding, or using mixed case, to bypass input filters and WAF rules.
- Test for IP and user agent blocking: Test if your WAF can block specific IPs or user agents, and check for bypass techniques using proxies or fake user agents.
- Test for rate limiting: Test if your WAF can enforce rate limiting and block requests that exceed the allowed rate.
- Test for cookie security: Test if your WAF can detect and block cookie manipulation, such as injecting malicious code or altering session cookies.
- Test for file upload vulnerabilities: Test if your WAF can detect and block malicious file uploads, such as uploading web shells or malware.
- Test for known attack signatures: Test your WAF's ability to detect and block known attack signatures using tools like Burp Suite or OWASP ZAP.
- Test custom WAF rules: Test custom WAF rules and configurations to ensure they properly block malicious requests.
- Test for false positives: Ensure your WAF doesn't block legitimate traffic by testing with common requests and inputs that may trigger false positives.
- Test for false negatives: Ensure your WAF doesn't allow malicious traffic by testing with known attack vectors that should trigger blocking.
- Test for SSL/TLS vulnerabilities: Test if your WAF can detect and block SSL/TLS vulnerabilities, such as POODLE or Heartbleed.
- Test for XML vulnerabilities: Test if your WAF can detect and block XML-based attacks, such as XXE or XEE.
- Test for header injection: Test if your WAF can detect and block header injection attacks, such as CRLF injection or response splitting.
- Test for path traversal attacks: Test if your WAF can detect and block path traversal attacks, such as directory traversal or file inclusion.
- Test for application-layer DDoS attacks: Test if your WAF can detect and block application-layer DDoS attacks, such as Slowloris or RUDY.
- Perform continuous testing and monitoring: Regularly test your WAF's effectiveness and monitor its logs to detect and block new attack vectors and emerging threats.

Header Vulnerability

- Missing Strict-Transport-Security (HSTS) header: Enables HTTPS-only communication, preventing man-in-the-middle attacks.
- Missing X-Content-Type-Options header: Disables MIME type sniffing, reducing the risk of attacks using MIME confusion.
- Missing X-Frame-Options header: Prevents clickjacking attacks by disallowing or limiting the site from being embedded within frames.
- Missing Content-Security-Policy (CSP) header: Defines allowed sources of content, reducing the risk of cross-site scripting (XSS) and content injection attacks.
- Missing X-XSS-Protection header: Activates built-in browser protection against cross-site scripting (XSS) attacks.
- Missing Referrer-Policy header: Controls the information sent in the Referer header, protecting user privacy and reducing the risk of information leakage.
- Missing Feature-Policy header: Restricts the use of certain browser features and APIs, improving security and privacy.
- Insecure CORS (Cross-Origin Resource Sharing) settings: Allows unauthorized domains to access resources, increasing the risk of cross-site request forgery (CSRF) and data leakage.
- Missing Expect-CT header: Enforces Certificate Transparency, reducing the risk of misissued SSL/TLS certificates.
- Missing Permissions-Policy header: Defines which browser features are allowed or denied, enhancing user privacy and security.
- Weak or missing Public-Key-Pins (HPKP) header: Ensures the use of specific cryptographic public keys, reducing the risk of man-in-the-middle attacks using rogue certificates.
- Missing X-Download-Options header: Prevents file download prompts from being displayed, reducing the risk of drive-by download attacks.
- Missing X-Permitted-Cross-Domain-Policies header: Restricts the loading of content from other domains, reducing the risk of data theft.
- Missing X-DNS-Prefetch-Control header: Controls DNS prefetching, potentially improving user privacy.
- Inadequate Cache-Control settings: Insecure caching settings can expose sensitive information or allow unauthorized access to content.
- Missing X-Content-Duration header: Helps prevent unauthorized media access by specifying the duration of media files.
- Missing Access-Control-Allow-Origin header: Improper configuration can result in unauthorized cross-origin resource sharing.
- Missing X-WebKit-CSP header: This older header is used by some legacy browsers for content security policy enforcement.
- Missing X-Content-Security-Policy header: Similar to X-WebKit-CSP, this older header is used by some legacy browsers for content security policy enforcement.
- Missing X-XContent-Type-Options header: Disables MIME sniffing on older browsers, reducing the risk of MIME confusion attacks.
- Insecure ETag settings: Weak ETag settings can cause caching issues, potentially exposing sensitive information.
- Missing or weak Content-Encoding header: Properly configuring this header helps protect against attacks that rely on manipulating content encoding.
- Missing or weak Content-Language header: Properly configuring this header helps protect against attacks that rely on manipulating content language.
- Missing or weak Last-Modified header: Properly configuring this header helps protect against attacks that rely on manipulating content modification timestamps.
- Insecure or missing Cookie headers: As mentioned in the previous answer, insecure cookie settings can lead to various security vulnerabilities.

SQL Injection

- Single quote test: Inject a single quote ' into input fields and observe if it generates an error or unexpected behavior, which might indicate a potential SQLi vulnerability.
- Tautologies: Inject tautologies like 1=1 or a=a into input fields or URL parameters to test for boolean-based SQLi.
- Union-based SQLi: Use the UNION operator to combine the results of two or more SELECT statements and extract data from other tables.
- Error-based SQLi: Inject incorrect syntax or invalid input to trigger error messages that reveal database structure or sensitive information.
- Time-based SQLi: Inject time-delaying functions like SLEEP() or WAITFOR DELAY to test for time-based SQLi vulnerabilities.
- Out-of-band (OOB) SQLi: Test for OOB SQLi by injecting payloads that cause the database to make external requests, such as DNS lookups or HTTP requests, to exfiltrate data.
- Double encoding: Test with double-encoded payloads to bypass filters that only decode input once. Example: %253Cscript%253Ealert(1)%253C%252Fscript%253E.
- Use SQL comment characters: Inject SQL comment characters (--, /*, */) to bypass input filters or terminate SQL statements.
- Manipulate query logic: Inject logical operators such as AND or OR to manipulate the query's logic and bypass access controls.
- Test with different SQL dialects: Use payloads specific to different SQL dialects (e.g., MySQL, PostgreSQL, Oracle, or MSSQL) to identify database-specific vulnerabilities.
- Test various HTTP methods: Test for SQLi vulnerabilities using different HTTP methods, such as POST, PUT, or PATCH, with SQLi payloads.
- Test with URL-encoded or base64-encoded parameters: Try URL-encoded or base64-encoded parameters to bypass input validation or access control checks.
- Test various content types: Test for SQLi vulnerabilities in different content types that support user input, such as JSON, XML, or URL-encoded form data.
- Manipulate cookies: Inject SQL payloads into cookies to test if the application processes them in an unsafe manner.
- Use web application scanners: Use automated web application scanners, such as Burp Suite or OWASP ZAP, to identify potential SQLi vulnerabilities.

TLS Vulnerability

- Weak or outdated SSL/TLS protocols: Ensure your site only supports secure and up-to-date protocols like TLS 1.2 and TLS 1.3, and disable insecure ones like SSL 2.0, SSL 3.0, and TLS 1.0.
- Insecure cipher suites: Disable weak cipher suites and use strong ones, such as those based on AES-GCM, ChaCha20-Poly1305, or ECDHE (Elliptic Curve Diffie-Hellman).
- Vulnerability to known attacks: Protect your site from known TLS attacks, such as POODLE, BEAST, CRIME, BREACH, or Heartbleed, by applying security patches and following best practices.
- Inadequate certificate management: Use a valid, trusted, and up-to-date SSL/TLS certificate from a reputable Certificate Authority (CA). Regularly check for certificate expiration and renewals.
- Insufficient certificate chain validation: Ensure proper validation of the certificate chain to prevent man-in-the-middle attacks using rogue or misissued certificates.
- Weak or missing public key pinning: Implement HTTP Public Key Pinning (HPKP) or Certificate Transparency to enforce the use of specific public keys and reduce the risk of man-in-the-middle attacks.
- Mixed content: Ensure that all content, including images, stylesheets, and scripts, are served over HTTPS to prevent mixed content warnings and potential attacks.
- Insecure renegotiation: Disable insecure client-initiated renegotiation to protect your site from man-in-the-middle attacks exploiting this vulnerability.
- Insufficient forward secrecy: Use cipher suites that support forward secrecy, such as ECDHE or DHE, to protect past communications from being decrypted even if the server's private key is compromised.
- Lack of OCSP stapling: Implement OCSP (Online Certificate Status Protocol) stapling to reduce the latency of SSL/TLS handshakes and provide real-time certificate revocation information.