Contents

1 WordPress Security Benchmark — DRAFT 1

1.1 Overview . . . 1
1.2 Target Technology . . . 2
1.3 Profile Definitions . . . 2
1.4 Assessment Status . . . 2
1.5 1 Web Server Configuration . . . 2
1.6 **References:** https://ssl-config.mozilla.org/ . . . 3
1.7 2 PHP Configuration . . . 6
1.8 3 Database Configuration . . . 9
1.9 4 WordPress Core Configuration . . . 12
1.10 5 Authentication and Access Control . . . 16
1.11 6 File System Permissions . . . 22
1.12 7 Logging and Monitoring . . . 24
1.13 8 Supply Chain and Extension Management . . . 26
1.14 9 Web Application Firewall . . . 29
1.15 10 Backup and Recovery . . . 29
1.16 11 AI and Generative AI Security . . . 30
1.17 12 Server Access and Network . . . 32
1.18 13 Multisite Security . . . 35
1.19 Appendix A: Recommendation Summary . . . 37
1.20 Related Documents . . . 38
1.21 License and Attribution . . . 38

1 WordPress Security Benchmark — DRAFT

Full Stack Hardening Guide

WordPress 6.x on Linux (Ubuntu/Debian)

Nginx or Apache • PHP 8.x • MySQL 8.x / MariaDB 10.x+

Dan Knauss February 16, 2026

1.1 Overview

This document provides prescriptive guidance for establishing a secure configuration posture for WordPress 6.x running on a Linux server stack. This benchmark covers the full stack: the operating system firewall, web server (Nginx or Apache), PHP runtime, MySQL/MariaDB database, and the WordPress application layer.

This benchmark is intended for system administrators, security engineers, DevOps teams, and WordPress developers responsible for deploying and maintaining WordPress installations in enterprise environments.

The guidance draws on many WordPress security resources and standards, such as the OWASP Top 10 (2025), NIST SP 800-63B, and field experience with enterprise WordPress hardening.

1.2 Target Technology

- WordPress 6.x (latest stable release recommended)
- Ubuntu 22.04+ / Debian 12+ (or equivalent RHEL/CentOS)
- Nginx 1.24+ or Apache 2.4+
- PHP 8.2+ (8.3+ recommended for new deployments)
- MySQL 8.0+ or MariaDB 10.6+

Note on Containerization: While this benchmark assumes a traditional Linux stack, the principles and many of the configuration settings apply equally to containerized environments (Docker, Kubernetes). In such cases, configurations should be injected via environment variables or secret management systems rather than direct file edits where possible.

1.3 Profile Definitions

This benchmark defines two configuration profiles:

Level	Description
Level 1	Essential security settings that can be implemented on any WordPress deployment with minimal impact on functionality or performance. These form a baseline security posture that every site should meet. Implementing Level 1 items should not significantly inhibit the usability of the technology.
Level 2	Defense-in-depth settings intended for high-security environments. These recommendations may restrict functionality, require additional tooling, or involve operational overhead. They are appropriate for sites handling sensitive data, regulated industries, or high-value targets.

1.4 Assessment Status

Automated: Compliance can be verified programmatically using command-line tools, configuration file inspection, or API queries.

Manual: Compliance requires human judgment, review of policies, or inspection of settings through a graphical interface.

1.5 1 Web Server Configuration

This section provides recommendations for hardening the web server (Nginx or Apache) that serves the WordPress application.

1.5.0.1 1.1 Ensure TLS 1.2+ is enforced Profile Applicability: Level 1

Assessment Status: Automated

Description: Only TLS 1.2 and TLS 1.3 should be accepted. TLS 1.0 and 1.1 contain known vulnerabilities and must be disabled.

Rationale: TLS 1.0 and 1.1 are vulnerable to BEAST, POODLE, and other attacks. All major browsers have dropped support for these protocols. Enforcing 1.2+ eliminates a class of protocol-level attacks.

Impact: Legacy clients that do not support TLS 1.2 will be unable to connect. This is an acceptable trade-off for security.

Audit:

For Nginx, verify the ssl_protocols directive:

```
$ grep -r 'ssl_protocols' /etc/nginx/
```

Verify that the output contains only TLSv1.2 and TLSv1.3. For Apache:

```
$ grep -r 'SSLProtocol' /etc/apache2/
```

Verify the output shows 'all -SSLv3 -TLSv1 -TLSv1.1' or equivalent.

Remediation:

For Nginx, set in the server or http block:

```
ssl_protocols TLSv1.2 TLSv1.3;
```

For Apache, set in the VirtualHost or global config:

```
SSLProtocol all -SSLv3 -TLSv1 -TLSv1.1
```

Restart the web server after changes.

Default Value: Nginx: TLSv1 TLSv1.1 TLSv1.2 (all enabled). Apache: All protocols enabled.

1.6 References: https://ssl-config.mozilla.org/

1.6.0.1 1.2 Ensure HTTP security headers are configured Profile Applicability: Level 1

Assessment Status: Automated

Description: The web server should send security-related HTTP headers including Content-Security-Policy, X-Content-Type-Options, X-Frame-Options, Strict-Transport-Security (HSTS), Referrer-Policy, and Permissions-Policy.

Rationale: HTTP security headers instruct the browser to enable built-in protections against common attacks such as XSS, clickjacking, MIME-type confusion, and insecure referrer leakage.

Impact: Overly restrictive Content-Security-Policy headers may break inline scripts, third-party integrations, or analytics tools. Note that `unsafe-inline` is often required for WordPress themes and plugins but represents a security trade-off. For Level 2, aim to remove `unsafe-inline` by using nonces or hashes.

Audit:

For Nginx, inspect the response headers:

```
$ curl -sI https://example.com | grep -iE '(content-security\|x-content-type\|x-frame\|strict-t
```

Verify all six headers are present.

Remediation:

For Nginx, add to the server block:

```
add_header X-Content-Type-Options "nosniff" always;

add_header X-Frame-Options "SAMEORIGIN" always;

add_header Referrer-Policy "strict-origin-when-cross-origin" always;

add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;

add_header Permissions-Policy "geolocation=(), camera=(), microphone=()" always;

add_header Content-Security-Policy "default-src 'self'; script-src 'self'; style-src 'self' 'un
```

For Apache, use the Headers module:

```
Header always set X-Content-Type-Options "nosniff"
Header always set X-Frame-Options "SAMEORIGIN"
```

Default Value: No security headers are set by default.

References: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers OWASP Secure Headers Project

1.6.0.2 1.3 Ensure server tokens and version information are hidden Profile Applicability: Level 1

Assessment Status: Automated

Description: The web server should not disclose its version number, operating system, or module information in HTTP response headers or error pages.

Rationale: Version information helps attackers identify specific vulnerabilities to target. Removing it forces attackers to probe the server more actively, increasing the chance of detection.

Audit:

```
$ curl -sI https://example.com | grep -i 'server'
```

Verify the Server header does not contain version numbers.

Remediation:

For Nginx:

```
server_tokens off;
```

For Apache:

```
ServerTokens Prod
ServerSignature Off
```

Default Value: Nginx: server_tokens on (version exposed). Apache: ServerTokens Full.

1.6.0.3 1.4 Ensure direct PHP execution is blocked in upload directories Profile Applicability: Level 1

Assessment Status: Automated

Description: PHP execution must be disabled in the wp-content/uploads/ directory and any other directories intended only for static file storage.

Rationale: If an attacker uploads a malicious PHP file through a vulnerability (e.g., an insecure file upload in a plugin), blocking PHP execution in the uploads directory prevents the file from being executed.

Impact: None. Legitimate WordPress operations never require PHP execution from the uploads directory.

Audit:

For Nginx, verify a location block exists for uploads:

```
$ grep -A5 'uploads' /etc/nginx/sites-enabled/\*
```

Verify that PHP processing is denied for the uploads directory.

Remediation:

For Nginx, add to the server block:

```
location ~* /wp-content/uploads/.*\.php$ {
  deny all;
}
```

For Apache, create wp-content/uploads/.htaccess:

```
<FilesMatch "\.php$">
  Require all denied
</FilesMatch>
```

Default Value: PHP execution is allowed in all directories by default.

References: https://developer.wordpress.org/advanced-administration/security/hardening/

1.6.0.4 1.5 Ensure rate limiting is configured for all API surfaces Profile Applicability: Level 1

Assessment Status: Automated

Description: HTTP request rate limiting should be applied to all authentication and API endpoints, including `wp-login.php`, `xmlrpc.php`, and the WordPress REST API (`/wp-json/`).

Rationale: WordPress authentication and API interfaces are primary targets for automated brute-force and resource exhaustion attacks. Comprehensive rate limiting across all entry points reduces the effectiveness of these attacks and protects server resources.

Impact: Aggressive rate limiting may lock out legitimate users or break third-party integrations (e.g., decoupled front-ends) if not configured with appropriate burst allowances and allowlist exceptions.

Audit:

For Nginx, check for `limit_req` configuration:

```
$ grep -r 'limit_req' /etc/nginx/
```

Verify rate limiting zones are defined and applied to login, XML-RPC, and REST API locations.

Remediation:

For Nginx, define rate limiting zones and apply them to the relevant locations:

```
# In http block:
limit_req_zone $binary_remote_addr zone=wplogin:10m rate=1r/s;
limit_req_zone $binary_remote_addr zone=wpapi:10m rate=5r/s;

# In server block:
location = /wp-login.php {
    limit_req zone=wplogin burst=3 nodelay;
    # ... PHP processing ...
}

location = /xmlrpc.php {
    limit_req zone=wplogin burst=3 nodelay;
    # ... PHP processing ...
}

location ~ ^/wp-json/ {
    limit_req zone=wpapi burst=10 nodelay;
    # ... PHP processing ...
}
```

Default Value: No rate limiting is configured by default.

1.7 2 PHP Configuration

This section provides recommendations for securing the PHP runtime environment.

1.7.0.1 2.1 Ensure expose_php is disabled Profile Applicability: Level 1

Assessment Status: Automated

Description: The expose_php directive in php.ini must be set to Off. This prevents PHP from disclosing its presence and version in HTTP response headers (X-Powered-By).

Rationale: Version information assists attackers in identifying vulnerabilities specific to the running PHP version.

Audit:

```
$ php -i | grep expose_php
```

Verify the output shows 'expose_php => Off => Off'.

Remediation:

In php.ini:

```
expose_php = Off
```

Restart PHP-FPM or the web server.

Default Value: expose_php = On

1.7.0.2 2.2 Ensure display_errors is disabled in production Profile Applicability: Level 1

Assessment Status: Automated

Description: The display_errors directive must be set to Off in production environments. PHP errors should be logged to a file, not displayed to users.

Rationale: Displayed PHP errors can reveal file paths, database connection details, and application structure to attackers.

Audit:

```
$ php -i | grep display_errors
```

Verify: 'display_errors => Off => Off'.

Remediation:

In php.ini:

```
display_errors = Off

log_errors = On

error_log = /var/log/php/error.log
```

Default Value: display_errors = On in development configurations.

1.7.0.3 2.3 Ensure dangerous PHP functions are disabled Profile Applicability: Level 1

Assessment Status: Automated

Description: PHP functions that allow arbitrary command execution, code evaluation, or information disclosure should be disabled unless specifically required.

Rationale: If an attacker achieves code execution (e.g., through a vulnerable plugin), these functions enable them to execute system commands, read arbitrary files, or escalate the attack.

Impact: Some WordPress plugins may require specific functions. Test thoroughly before deploying. The `eval()` function is a language construct and cannot be disabled via `disable_functions`.

Recommendation (Level 2): For high-security environments, consider using a PHP security extension like **Snuffleupagus** to mitigate `eval()` and provide additional hardening that `disable_functions` cannot achieve.

Audit:

```
$ php -i | grep disable_functions
```

Verify the output includes dangerous functions.

Remediation:

In php.ini:

```
disable_functions = exec,passthru,shell_exec,system,proc_open,popen,curl_multi_exec,parse_ini_
```

Default Value: No functions are disabled by default.

1.7.0.4 2.4 Ensure open_basedir restricts file access Profile Applicability: Level 2

Assessment Status: Automated

Description: The open_basedir directive should restrict PHP file operations to the WordPress installation directory and required system paths only.

Rationale: open_basedir prevents PHP code from reading or writing files outside the defined directory tree, limiting the impact of a file inclusion or traversal vulnerability.

Impact: Must include the WordPress root, /tmp (for file uploads), and the PHP session directory. Incorrect configuration will break WordPress.

Audit:

```
$ php -i | grep open_basedir
```

Verify a restricted path is configured.

Remediation:

In the PHP-FPM pool configuration or php.ini:

```
open_basedir = /var/www/example.com:/tmp:/usr/share/php
```

Default Value: open_basedir is not set (unrestricted).

1.7.0.5 2.5 Ensure PHP session security is configured Profile Applicability: Level 1

Assessment Status: Automated

Description: PHP session configuration should enforce secure defaults: cookies marked Secure, HttpOnly, and SameSite=Lax or Strict. Session ID entropy and hashing should use strong algorithms.

Rationale: Secure session configuration prevents session fixation, cookie theft via XSS, and cross-site request forgery via session cookies.

Audit:

```
$ php -i | grep -E 'session\.(cookie_secure\|cookie_httponly\|cookie_samesite\|use_strict_mode)
```

Verify all are set to appropriate secure values.

Remediation:

In php.ini:

```
session.cookie_secure = 1

session.cookie_httponly = 1

session.cookie_samesite = Lax

session.use_strict_mode = 1

session.use_only_cookies = 1
```

Default Value: session.cookie_secure = 0, session.cookie_httponly = 0 (insecure defaults).

References: https://www.php.net/manual/en/session.security.ini.php

---

1.8 3 Database Configuration

This section covers MySQL/MariaDB configuration relevant to WordPress security.

1.8.0.1 3.1 Ensure the WordPress database user has minimal privileges Profile Applicability: Level 1

Assessment Status: Automated

Description: The MySQL/MariaDB user account used by WordPress should have only the privileges required for normal operation: SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, INDEX, and DROP on the WordPress database only.

Rationale: Granting excessive privileges (e.g., FILE, SUPER, GRANT) increases the impact of a SQL injection vulnerability. With minimal privileges, an attacker who achieves SQLi cannot read arbitrary files, modify grants, or perform administrative operations.

Impact: Some plugins may require CREATE TEMPORARY TABLES or LOCK TABLES. Add only when verified necessary.

Audit:

Run as the MySQL root user:

```
SELECT user, host FROM mysql.user;
```

```
SHOW GRANTS FOR 'wp_user'@'localhost';
```

Verify the user has privileges only on the WordPress database and only the required types.

Remediation:

```
REVOKE ALL PRIVILEGES ON *.* FROM 'wp_user'@'localhost';
```

```
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, INDEX, DROP ON wp_database.* TO 'wp_user'
```

```
FLUSH PRIVILEGES;
```

Default Value: Depends on initial setup. Many installation guides grant ALL PRIVILEGES.

1.8.0.2 3.2 Ensure the database is not accessible from external hosts Profile Applicability: Level 1

Assessment Status: Automated

Description: MySQL/MariaDB should be configured to listen only on localhost (127.0.0.1) or a Unix socket. Remote TCP connections should be disabled unless required and tunneled through SSH or a VPN.

Rationale: A database accessible over the network expands the attack surface. Brute-force attacks, credential stuffing, and exploitation of database vulnerabilities become possible from any host that can reach the port.

Audit:

```
$ grep -E 'bind-address\|skip-networking' /etc/mysql/mysql.conf.d/mysqld.cnf
```

Verify bind-address is 127.0.0.1 or ::1.

```
$ ss -tlnp | grep 3306
```

Verify MySQL is listening only on 127.0.0.1:3306.

Remediation:

In mysqld.cnf or my.cnf under [mysqld]:

```
bind-address = 127.0.0.1
```

Restart MySQL.

Default Value: bind-address = 0.0.0.0 (listening on all interfaces) on some distributions.

1.8.0.3 3.3 Ensure a non-default table prefix is used Profile Applicability: Level 1

Assessment Status: Manual

Description: WordPress should be configured with a database table prefix other than the default wp_ to make automated SQL injection attacks less effective.

Rationale: Automated attack tools typically assume the default wp_ prefix. A non-default prefix requires attackers to discover the actual table names, adding a layer of difficulty.

Impact: Changing the prefix on an existing installation requires updating all table names and option/usermeta values that reference the prefix. This is best done at installation time.

Audit:

Inspect `wp-config.php`:

```
$ grep 'table_prefix' /path/to/wp-config.php
```

Verify the value is not 'wp_'.

Remediation:

In `wp-config.php`, set during installation:

```
$table_prefix = 'wxyz_';
```

Use a short, random string. Do not use personally identifiable or guessable values.

Default Value: $table_prefix = 'wp_';

1.8.0.4 3.4 Ensure database query logging is enabled Profile Applicability: Level 2

Assessment Status: Automated

Description: MySQL/MariaDB general query log or slow query log should be enabled to support forensic analysis and intrusion detection.

Rationale: Query logs provide critical evidence during incident response, including the exact queries executed by an attacker who achieved SQL injection. They also help identify performance issues that may indicate abuse.

Impact: General query logging incurs significant I/O overhead and should be used selectively or only during investigations. Slow query logging has minimal overhead and can remain enabled.

Audit:

```
$ grep -E '(general_log\|slow_query_log)' /etc/mysql/mysql.conf.d/mysqld.cnf
```

Verify at minimum slow_query_log is enabled.

Remediation:

In mysqld.cnf under [mysqld]:

```
slow_query_log = 1
```

```
slow_query_log_file = /var/log/mysql/mysql-slow.log
```

```
long_query_time = 2
```

For investigations, temporarily enable:

```
general_log = 1
```

```
general_log_file = /var/log/mysql/mysql-general.log
```

Default Value: Both logs are disabled by default.

1.9 4 WordPress Core Configuration

This section covers security settings in `wp-config.php` and WordPress core behavior.

1.9.0.1 4.1 Ensure `DISALLOW_FILE_MODS` is set to true Profile Applicability: Level 1

Assessment Status: Automated

Description: The `DISALLOW_FILE_MODS` constant should be defined as `true` in `wp-config.php`. This prevents all file modifications through the WordPress admin interface, including plugin and theme installation, updates, and code editing.

Rationale: If an attacker gains admin access (e.g., through a compromised account), `DISALLOW_FILE_MODS` prevents them from installing malicious plugins, modifying theme files, or uploading web shells through the admin interface. Updates should be handled through deployment pipelines or server-side automation.

Impact: Plugin and theme updates cannot be performed through the Dashboard. An alternative update mechanism (`wp-cli`, CI/CD pipeline, or managed hosting) is required.

Audit:

```
$ grep 'DISALLOW_FILE_MODS' /path/to/wp-config.php
```

Verify: `define( 'DISALLOW_FILE_MODS', true );`

Remediation:

Add to `wp-config.php` before ‘That’s all, stop editing!’: `define( 'DISALLOW_FILE_MODS', true );`

Default Value: Not set (file modifications allowed).

References: https://developer.wordpress.org/advanced-administration/security/hardening/

1.9.0.2 4.2 Ensure `FORCE_SSL_ADMIN` is set to true Profile Applicability: Level 1

Assessment Status: Automated

Description: The `FORCE_SSL_ADMIN` constant forces all admin and login pages to be served over HTTPS.

Rationale: Without this setting, admin session cookies could be transmitted over unencrypted HTTP if a user accesses the admin via an HTTP URL, enabling session hijacking via network interception.

Audit:

```
$ grep 'FORCE_SSL_ADMIN' /path/to/wp-config.php
```

Verify: `define( 'FORCE_SSL_ADMIN', true );`

Remediation:

Add to `wp-config.php`: `define( 'FORCE_SSL_ADMIN', true );`

Default Value: Not set (HTTPS not enforced for admin).

1.9.0.3 4.3 Ensure WordPress debug mode is disabled in production Profile Applicability: Level 1

Assessment Status: Automated

Description: `WP_DEBUG` must be set to `false` in production environments. `WP_DEBUG_DISPLAY` must also be `false`, and `WP_DEBUG_LOG` should write to a non-public location if enabled.

Rationale: Debug output can reveal file paths, database queries, and PHP errors to attackers. Debug log files in the default location (`wp-content/debug.log`) are publicly accessible unless explicitly blocked.

Audit:

```
$ grep -E 'WP_DEBUG\|WP_DEBUG_DISPLAY\|WP_DEBUG_LOG' /path/to/wp-config.php
```

Verify `WP_DEBUG` and `WP_DEBUG_DISPLAY` are `false`. If `WP_DEBUG_LOG` is enabled, verify the log path is outside the web root or blocked by the web server.

Remediation:

`define( 'WP_DEBUG', false );`

`define( 'WP_DEBUG_DISPLAY', false );`

`define( 'WP_DEBUG_LOG', false );` If logging is needed, direct to a non-public path: `define( 'WP_DEBUG_LOG', '/var/log/wordpress/debug.log' );`

Default Value: `WP_DEBUG = false` (secure by default). However, many deployment guides enable debug mode.

1.9.0.4 4.4 Ensure XML-RPC is disabled Profile Applicability: Level 1

Assessment Status: Automated

Description: The XML-RPC interface (`xmlrpc.php`) should be disabled unless specifically required by a remote publishing client or integration.

Rationale: XML-RPC is commonly exploited for brute-force amplification attacks (the `system.multicall` method allows hundreds of password attempts in a single HTTP request) and DDoS amplification via pingbacks.

Impact: Disabling XML-RPC will break Jetpack (which requires it for WordPress.com communication), the WordPress mobile app (older versions), and any third-party tool that uses the XML-RPC API.

Audit:

```
$ curl -s -o /dev/null -w '%{http_code}' https://example.com/xmlrpc.php
```

A 200 response indicates XML-RPC is accessible. A 403 or 404 indicates it is blocked.

Remediation:

Block at the web server level (preferred). For Nginx:

```
location = /xmlrpc.php {
  deny all;
  return 403;
}
```

Or disable via `wp-config.php`: `add_filter( 'xmlrpc_enabled', '__return_false' );` (Place in a must-use plugin, not `wp-config.php` directly.)

Additionally, disable trackbacks and pingbacks in **Settings → Discussion** by unchecking "Allow link notifications from other blogs (pingbacks and trackbacks) on new posts." Trackbacks operate independently of `xmlrpc.php` and should be disabled separately.

Default Value: XML-RPC, trackbacks, and pingbacks are all enabled by default.

References: https://developer.wordpress.org/advanced-administration/security/hardening/

1.9.0.5 4.5 Ensure automatic core updates are enabled Profile Applicability: Level 1

Assessment Status: Automated

Description: WordPress automatic background updates for minor (security) releases must remain enabled. Major version auto-updates should be evaluated based on organizational policy.

Rationale: Minor releases contain only security fixes and critical bug patches. Disabling them leaves the site vulnerable to known, publicly disclosed exploits.

Impact: In rare cases, a minor update may introduce a regression. Managed hosting providers typically handle this with rollback capabilities.

Audit:

```
$ wp config get WP_AUTO_UPDATE_CORE --path=/path/to/wordpress 2>/dev/null
```

```
$ grep 'WP_AUTO_UPDATE_CORE\|AUTOMATIC_UPDATER_DISABLED' /path/to/wp-config.php
```

Verify `WP_AUTO_UPDATE_CORE` is not set to `false` and `AUTOMATIC_UPDATER_DISABLED` is not `true`.

Remediation:

Ensure `wp-config.php` does not contain: `define( 'AUTOMATIC_UPDATER_DISABLED', true );` Optionally, explicitly enable minor updates: `define( 'WP_AUTO_UPDATE_CORE', 'minor' );`

Default Value: Minor auto-updates are enabled by default since WordPress 3.7.

1.9.0.6 4.6 Ensure unique authentication keys and salts are configured Profile Applicability: Level 1

Assessment Status: Automated

Description: All eight authentication keys and salts in `wp-config.php` must be set to unique, random values. These are: `AUTH_KEY`, `SECURE_AUTH_KEY`, `LOGGED_IN_KEY`, `NONCE_KEY`, and their corresponding `SALT` counterparts.

Rationale: These keys are used to hash session tokens stored in cookies. Default, empty, or guessable values weaken cookie security, making session forgery and hijacking easier.

Audit:

```
$ grep -E '(AUTH_KEY\|SECURE_AUTH_KEY\|LOGGED_IN_KEY\|NONCE_KEY\|AUTH_SALT\|SECURE_AUTH_SALT\|
```

Verify all eight constants are defined with long, unique random strings. None should be 'put your unique phrase here' (the placeholder value).

Remediation:

Generate new keys using the WordPress.org API:

```
$ curl -s https://api.wordpress.org/secret-key/1.1/salt/
```

Replace the key definitions in `wp-config.php` with the generated output.

Default Value: Placeholder values ('put your unique phrase here') in fresh installations.

References: https://developer.wordpress.org/advanced-administration/security/hardening/

1.9.0.7 4.7 Ensure `wp-cron.php` is replaced with a system cron job Profile Applicability: Level 1

Assessment Status: Automated

Description: WordPress's built-in pseudo-cron (`wp-cron.php`) should be disabled and replaced with a real system-level cron job. `wp-cron.php` is triggered on every page load, making execution timing unpredictable and exposing an additional PHP endpoint to the network.

Rationale: `wp-cron.php` can be abused for resource exhaustion by sending rapid requests to the endpoint. It is also unreliable on low-traffic sites where scheduled tasks may not fire at all. A system cron job executes on a predictable schedule regardless of traffic and eliminates an unnecessary public PHP endpoint.

Impact: Disabling `wp-cron.php` without configuring a system cron replacement will prevent scheduled tasks (e.g., publishing scheduled posts, checking for updates, sending email digests) from running.

Audit:

```
$ grep 'DISABLE_WP_CRON' /path/to/wp-config.php
```

Verify: `define( 'DISABLE_WP_CRON', true );`

Verify a system cron job is configured:

```
$ crontab -l | grep wp-cron
```

Remediation:

1. Add to `wp-config.php`: `define( 'DISABLE_WP_CRON', true );`
2. Add a system cron job (runs every 5 minutes) using `wp-cli`:

```
*/5 * * * * cd /path/to/wordpress && wp cron event run --due-now > /dev/null 2>&1
```

3. Block direct external access to `wp-cron.php` at the web server level (Nginx):

```
location = /wp-cron.php {
    deny all;
    return 403;
}
```

This is safe because `wp-cli` executes PHP directly and does not use HTTP.

Default Value: `wp-cron.php` is enabled and triggered on every page load by default.

1.10 5 Authentication and Access Control

This section addresses user authentication, session management, and role-based access control within WordPress.

1.10.0.1 5.1 Ensure two-factor authentication is required for administrators Profile Applicability: Level 1

Assessment Status: Manual

Description: All user accounts with the Administrator role must have two-factor authentication (2FA) enabled using TOTP-based authenticator apps or hardware security keys (WebAuthn/FIDO2).

Rationale: Compromised administrator credentials grant full control over the WordPress installation. 2FA ensures that a stolen password alone is insufficient to gain access.

Impact: Requires a 2FA plugin (e.g., Two Factor, Wordfence, WP 2FA, or Fortress). WordPress core does not include 2FA natively as of version 6.9.

Audit:

This is a manual check. Verify that: 1. A 2FA plugin is installed and active. 2. All administrator accounts have 2FA configured. 3. SMS-based 2FA is not used (vulnerable to SIM-swapping).

Remediation:

Install and configure a 2FA plugin. Require 2FA enrollment for all users with Administrator, Editor, or Shop Manager roles. Recommended: Enforce 2FA as mandatory for admin roles with a grace period for initial setup.

Default Value: No 2FA is configured by default.

References: NIST SP 800-63B https://developer.wordpress.org/advanced-administration/security/hardening/

1.10.0.2 5.2 Ensure the number of administrator accounts is minimized Profile Applicability: Level 1

Assessment Status: Manual

Description: The number of user accounts with the Administrator role should be limited to the minimum required. A primary administrator account should be reserved for emergency use only.

Rationale: Each administrator account is a potential entry point. Compromising any single admin account grants full site control. Minimizing admin accounts reduces the attack surface.

Audit:

```
$ wp user list --role=administrator --fields=ID,user_login,user_email --path=/path/to/wordpress
```

Review the list. Verify that each admin account is actively needed and assigned to a specific individual.

Remediation:

1. Audit existing administrator accounts. 2. Downgrade accounts that don't require full admin capabilities to Editor or a custom role. 3. Reserve one primary administrator account for break-glass emergencies. 4. Use custom roles with tailored capabilities for day-to-day operations.

Default Value: One administrator account is created during installation.

1.10.0.3 5.3 Ensure maximum session lifetime is enforced Profile Applicability: Level 1

Assessment Status: Automated

Description: WordPress session cookies should have a maximum lifetime enforced, regardless of user activity. Privileged accounts (Administrators, Editors) should have shorter session limits (8–24 hours). Additionally, idle sessions should be terminated after a defined period of inactivity, the "Remember Me" option should be disabled or minimized for administrator accounts, and all active sessions should be purged on role or permission changes.

Rationale: Long-lived sessions increase the window of opportunity for session hijacking. If an auth cookie is stolen, a shorter lifetime limits how long the attacker can use it. Idle session timeouts and scheduled session destruction provide additional layers of defense that reduce the exposure window even further.

Impact: Users will need to re-authenticate more frequently. This can be mitigated with trusted device verification.

Audit:

Check for session management plugins or custom code:

```
$ grep -r 'auth_cookie_expiration' /path/to/wp-content/mu-plugins/ /path/to/wp-config.php
```

Verify a filter is in place to limit session lifetime.

Remediation:

Add a must-use plugin (wp-content/mu-plugins/session-limits.php):

```
add_filter( 'auth_cookie_expiration', function( $expiration, $user_id, $remember ) {
  $user = get_userdata( $user_id );

  if ( in_array( 'administrator', $user->roles ) ) {
    return 8 * HOUR_IN_SECONDS; // 8 hours for admins
  }

  return 24 * HOUR_IN_SECONDS; // 24 hours for others
}, 10, 3 );
```

Default Value: 48 hours (2 days) without 'Remember Me'; 14 days with 'Remember Me'.

1.10.0.4 5.4 Ensure user enumeration is prevented Profile Applicability: Level 1

Assessment Status: Automated

Description: The REST API user endpoint and author archive URLs should be restricted to prevent unauthenticated enumeration of usernames.

Rationale: Username enumeration provides attackers with valid login targets for brute-force and credential stuffing attacks. The default WordPress REST API exposes user slugs at `/wp-json/wp/v2/users`, and author archives expose usernames via `/?author=N` URLs.

Impact: Blocking the REST API users endpoint may affect plugins that rely on it for public author data (e.g., some theme author bio features).

Audit:

```
$ curl -s https://example.com/wp-json/wp/v2/users | python3 -m json.tool
```

If the response returns user data, enumeration is possible.

```
$ curl -sI https://example.com/?author=1
```

If the response is a 301 redirect to an author archive, enumeration is possible.

Remediation:

Block the REST API users endpoint for unauthenticated requests via a must-use plugin:

```
add_filter( 'rest_endpoints', function( $endpoints ) {
    if ( ! is_user_logged_in() ) {
        unset( $endpoints['/wp/v2/users'] );
        unset( $endpoints['/wp/v2/users/(?P<id>[\d]+)'] );
    }
    return $endpoints;
});
```

Block author archive enumeration at the web server level or with a plugin.

Default Value: User data is publicly accessible via the REST API and author archives.

1.10.0.5 5.5 Ensure reauthentication is required for privileged actions Profile Applicability: Level 2

Assessment Status: Manual

Description: WordPress should require reauthentication (sudo mode) before performing sensitive administrative actions. An "action-gated" model should be adopted to challenge users for their password (and 2FA) when they attempt destructive or high-risk operations.

Rationale: If a session is hijacked, reauthentication limits the damage the attacker can do with the stolen session. Gating critical operations ensures that even with a stolen browser cookie, the attacker cannot perform permanent or high-impact changes without knowing the user's password.

Impact: Requires a dedicated security solution (e.g., Fortress by Snicco). WordPress core does not natively enforce reauthentication for most admin actions.

Audit:

This is a manual check. Verify that a reauthentication challenge is triggered for at least the following categories of actions: 1. **Plugins & Themes:** Installation, deletion, and updates. 2. **Users:** Creation, deletion, and role promotion (especially to Administrator). 3. **Authentication:** Application password creation (to prevent persistent backdoors). 4. **Configuration:** Edits to `wp-config.php` constants or critical site options. 5. **Tools:** Data export (WXR) and WordPress core updates.

Remediation:

Implement an action-gated reauthentication solution. Configure the "Action Registry" to gate all high-priority destructive operations. Ensure the solution supports both the Dashboard UI and relevant API surfaces (AJAX/REST).

Default Value: WordPress requires password confirmation only for profile email/password changes.

1.10.0.6 5.6 Ensure unauthenticated REST API access is restricted Profile Applicability: Level 2

Assessment Status: Automated

Description: The WordPress REST API should be restricted to authenticated users only, except for specific public endpoints that require unauthenticated access (e.g., for front-end search or decoupled front-ends).

Rationale: By default, the REST API is open and provides significant information about the site structure, content, and users. Restricting access reduces the attack surface and prevents information leakage to unauthenticated actors.

Impact: Will break decoupled (headless) installations or plugins that rely on unauthenticated REST API access for front-end functionality.

Audit:

```
$ curl -sI https://example.com/wp-json/wp/v2/posts
```

If the response is `200 OK`, unauthenticated access is allowed. A `401 Unauthorized` or `403 Forbidden` indicates restricted access.

Remediation:

Add a must-use plugin:

```
add_filter( 'rest_authentication_errors', function( $result ) {
    if ( ! empty( $result ) ) {
        return $result;
    }
    if ( ! is_user_logged_in() ) {
        return new WP_Error( 'rest_not_logged_in', 'You are not currently logged in.', array(
    }
    return $result;
});
```

Default Value: REST API is accessible to unauthenticated users.

1.10.0.7 5.7 Ensure a strong password policy is enforced Profile Applicability: Level 1

Assessment Status: Manual

Description: Configure WordPress to enforce a strong password policy that follows current OWASP recommendations: minimum length of 12 characters, checking against breached password/dictionary lists, and avoiding arbitrary complexity rules that lead to predictable patterns.

Rationale: Weak passwords are a primary vector for account takeover. Modern standards (NIST SP 800-63B and OWASP) emphasize length and entropy over complexity (e.g., forcing special characters), and mandate checking against known compromised credentials.

Impact: Users may need to update existing weak passwords. Requires a plugin for advanced enforcement (e.g., Wordfence, iThemes Security, or Passthrough Authentication).

Note on Password Hashing: As of WordPress 6.8, user passwords are hashed with bcrypt by default. Argon2id is supported on compatible PHP environments and provides stronger resistance to GPU-accelerated brute-force attacks. For high-security deployments, consider enabling Argon2id via the `wp_hash_password` core filter, typically implemented as a must-use plugin.

Audit:

This is a manual check. Verify that: 1. A password enforcement mechanism is active. 2. Test by attempting to set a simple 8-character password; verify it is rejected. 3. Verify the policy requires at least 12 characters.

Remediation:

1. Install a security plugin that supports password policy enforcement. 2. Configure the policy to require a minimum of 12 characters. 3. Enable "pwned password" checks to block credentials found

in previous data breaches. 4. Remove legacy requirements for symbols or numbers if they interfere with user-generated passphrases.

Default Value: WordPress encourages strong passwords but does not strictly enforce a minimum length or check against breached lists by default.

References: OWASP Authentication Cheat Sheet, NIST SP 800-63B

1.10.0.8 5.8 Ensure user roles and capabilities are defined in code Profile Applicability: Level 2

Assessment Status: Manual

Description: User roles and custom capabilities should be defined in code (via a must-use plugin or `wp-config.php`) rather than relying solely on database-stored role definitions. This ensures role definitions are version-controlled, auditable, and resistant to tampering.

Rationale: Role and capability definitions stored only in the database can be modified by an attacker who achieves SQL injection or gains admin access. Defining roles in code makes privilege escalation via database manipulation significantly harder and ensures role definitions can be reviewed in version control.

Impact: Requires development effort to codify custom roles. Changes to roles must go through the deployment pipeline rather than the WordPress Dashboard.

Audit:

This is a manual check. Verify that: 1. Custom roles are defined in a must-use plugin or `wp-config.php`-adjacent include file. 2. Role definitions are stored in version control. 3. Default role capabilities have been reviewed and unnecessary capabilities removed.

Remediation:

Create a must-use plugin (`wp-content/mu-plugins/custom-roles.php`) that registers custom roles and removes unnecessary default capabilities on every load:

```
add_action( 'init', function() {
    // Remove capabilities from default roles as needed
    $editor = get_role( 'editor' );
    if ( $editor ) {
        $editor->remove_cap( 'unfiltered_html' );
    }

    // Register custom roles
    if ( ! get_role( 'site_manager' ) ) {
        add_role( 'site_manager', 'Site Manager', array(
            'read'           => true,
            'manage_options' => true,
            // Add only the capabilities this role requires
        ));
    }
});
```

Default Value: Roles are stored in the `wp_options` table and editable via plugins or direct database access.

1.11 6 File System Permissions

This section covers file ownership and permission settings for the WordPress installation.

1.11.0.1 6.1 Ensure WordPress files are owned by a non-web-server user Profile Applicability: Level 1

Assessment Status: Automated

Description: WordPress files should be owned by a system user account, not the web server process user (www-data, nginx, apache). The web server should have read access only, with write access limited to specific directories (uploads, cache).

Rationale: If the web server process is compromised, file ownership by a separate user prevents the attacker from modifying WordPress core, plugin, or theme files.

Impact: WordPress auto-updates and plugin installations via the Dashboard require write access to the file system. With DISALLOW_FILE_MODS enabled (see 4.1), this is not needed.

Audit:

`$ stat -c '%U:%G' /path/to/wordpress/wp-config.php`

`$ stat -c '%U:%G' /path/to/wordpress/wp-includes/version.php`

Verify files are not owned by www-data, nginx, or apache.

Remediation:

`sudo chown -R wp_user:www-data /path/to/wordpress/`

`sudo find /path/to/wordpress/ -type d -exec chmod 750 {} \;`

`sudo find /path/to/wordpress/ -type f -exec chmod 640 {} \;`

`sudo chmod 640 /path/to/wordpress/wp-config.php`

```
# Allow web server to write to uploads (if needed)
sudo find /path/to/wordpress/wp-content/uploads -type d -exec chmod 775 {} \;
sudo find /path/to/wordpress/wp-content/uploads -type f -exec chmod 664 {} \;
```

Default Value: Ownership depends on installation method. Many guides set www-data as owner.

1.11.0.2 6.2 Ensure `wp-config.php` has restrictive permissions Profile Applicability: Level 1

Assessment Status: Automated

Description: `wp-config.php` must have file permissions of 600 (owner read/write only) or 640 (owner read/write, group read). It should not be readable by the web server user directly; access should be through the PHP-FPM pool running as the site user.

Rationale: `wp-config.php` contains database credentials, authentication keys, and security-sensitive configuration. Broad read permissions could expose these to other users on a shared server or to a compromised web server process.

Audit:

```
$ stat -c '%a %U:%G' /path/to/wordpress/wp-config.php
```

Verify permissions are 600 or 640, and the owner is not the web server user.

Remediation:

```
chmod 600 /path/to/wordpress/wp-config.php
```

```
chown wp_user:wp_user /path/to/wordpress/wp-config.php
```

Default Value: 644 (world-readable) in many default configurations.

1.11.0.3 6.3 Ensure `wp-config.php` is placed above the document root Profile Applicability: Level 2

Assessment Status: Manual

Description: Where server configuration allows, `wp-config.php` should be moved one directory above the web document root. WordPress automatically detects this placement and loads the file from the parent directory.

Rationale: Placing `wp-config.php` above the document root prevents direct HTTP access to the file entirely, even if a web server misconfiguration exposes PHP source code. This provides defense-in-depth beyond file permissions alone.

Impact: Some hosting environments (e.g., shared hosting with restricted directory structures, some containerized setups) may not support this configuration. Additionally, if the WordPress installation is in the web root itself (rather than a subdirectory), the parent directory must not be another site's web root.

Audit:

Verify `wp-config.php` is not inside the document root:

```
$ ls -la /var/www/example.com/wp-config.php
```

If the file exists in the document root, it should be moved. Verify WordPress functions correctly after the move:

```
$ curl -sI https://example.com/ | head -5
```

Remediation:

Move `wp-config.php` one directory above the document root:

```
$ mv /var/www/example.com/wp-config.php /var/www/wp-config.php
```

WordPress will automatically detect and load `wp-config.php` from the parent directory. No additional configuration is needed.

Verify the parent directory is not publicly accessible and has restrictive permissions:

```
$ chmod 750 /var/www/
```

Default Value: `wp-config.php` is placed in the WordPress installation root (typically the document root) by default.

1.12 7 Logging and Monitoring

This section addresses audit logging, activity monitoring, and intrusion detection for WordPress.

1.12.0.1 7.1 Ensure WordPress user activity logging is enabled Profile Applicability: Level 1

Assessment Status: Manual

Description: A WordPress audit logging solution must be installed and configured to record all user activity, including: logins and logouts, failed login attempts, content creation and modification, user account changes, plugin and theme changes, and settings modifications.

Rationale: Audit logs are essential for detecting unauthorized activity, supporting incident response, and meeting compliance requirements (PCI DSS, HIPAA, GDPR). Without logging, security incidents may go undetected and forensic analysis is impossible.

Impact: Requires a third-party plugin. Recommended: WP Activity Log, Stream, or equivalent.

Audit:

This is a manual check. Verify that: 1. An audit logging plugin is installed and active. 2. Logs capture login activity, content changes, and settings modifications. 3. Logs are retained for a period consistent with compliance requirements.

Remediation:

Install and configure an audit logging plugin (e.g., WP Activity Log). Configure log retention for at least 90 days (or per organizational policy). Enable email alerts for critical events: failed logins, new admin users, plugin changes. Export logs to a centralized SIEM for correlation (Level 2).

Default Value: No audit logging is configured by default.

1.12.0.2 7.2 Ensure file integrity monitoring is configured Profile Applicability: Level 2

Assessment Status: Automated

Description: A mechanism should be in place to detect unauthorized changes to WordPress core files, plugins, themes, and configuration files.

Rationale: Unauthorized file modifications are a strong indicator of compromise. Integrity monitoring detects web shells, backdoors, and unauthorized code changes.

Impact: Can be implemented at the server level (AIDE, OSSEC, Tripwire) or WordPress level (Wordfence, Sucuri) or both.

Audit:

For WordPress core integrity:

```
$ wp core verify-checksums --path=/path/to/wordpress
```

For plugin integrity:

```
$ wp plugin verify-checksums --all --path=/path/to/wordpress
```

Verify both commands report no modifications.

Remediation:

1. Run `wp core verify-checksums` and `wp plugin verify-checksums` on a scheduled basis (daily recommended). 2. Install a file integrity monitoring plugin or configure server-level monitoring. 3. Alert on any unexpected file changes in `wp-includes/`, `wp-admin/`, and plugin directories.

Default Value: No integrity monitoring is configured by default.

References: https://developer.wordpress.org/cli/commands/core/verify-checksums/

1.12.0.3 7.3 Ensure server-level malware detection is configured Profile Applicability: Level 2

Assessment Status: Manual

Description: A server-level malware detection solution should be deployed to scan the WordPress file system for known malicious patterns, web shells, backdoors, and unauthorized modifications beyond what WordPress-level integrity checks can detect.

Rationale: WordPress-level file integrity checks (7.2) compare files against known-good checksums but cannot detect malware injected into non-core files, uploaded web shells, or obfuscated payloads in legitimate-looking files. Server-level scanning tools use signature databases and heuristic analysis that cover a broader threat surface.

Impact: Scanning may consume server resources. Schedule intensive scans during low-traffic periods. Real-time monitoring (where available) adds minimal overhead.

Audit:

This is a manual check. Verify that: 1. A server-level malware scanning tool is installed and active (e.g., Imunify360, Linux Malware Detect, ClamAV). 2. Regular scans are scheduled (daily recommended). 3. Scan results are reviewed and alerts are configured for detections.

Remediation:

1. Install a server-level malware detection tool appropriate to the environment:

   - **Managed hosting:** Verify the hosting provider includes malware scanning (most enterprise WordPress hosts include Imunify360 or equivalent).
   - **Self-managed:** Install Linux Malware Detect (LMD) with ClamAV as a scanning engine.
2. Configure daily scheduled scans of the WordPress installation directory.
3. Enable real-time monitoring of the `wp-content/` directory if supported.
4. Configure email or SIEM alerts for malware detections.

Default Value: No malware detection is configured by default on most server environments.

1.13 8 Supply Chain and Extension Management

This section addresses the security of WordPress plugins, themes, and their update processes.

1.13.0.1 8.1 Ensure all unused plugins and themes are removed Profile Applicability: Level 1

Assessment Status: Automated

Description: All deactivated plugins and non-active themes (except one default fallback theme) should be deleted from the server, not merely deactivated.

Rationale: Deactivated plugins and themes remain on the file system and may contain exploitable vulnerabilities. PHP files in deactivated plugins can be accessed directly if the web server processes them, bypassing WordPress entirely.

Audit:

```
$ wp plugin list --status=inactive --fields=name,version --path=/path/to/wordpress
```

```
$ wp theme list --status=inactive --fields=name,version --path=/path/to/wordpress
```

Verify no unused plugins or themes are present (one default/fallback theme is acceptable).

Remediation:

```
$ wp plugin delete <plugin-name> --path=/path/to/wordpress
```

```
$ wp theme delete <theme-name> --path=/path/to/wordpress
```

Retain only the active theme and one default WordPress theme as a fallback.

Default Value: Default themes and example plugins (Akismet, Hello Dolly) are included in fresh installations.

1.13.0.2 8.2 Ensure all plugins and themes are from trusted sources Profile Applicability: Level 1

Assessment Status: Manual

Description: Plugins and themes should only be installed from the official WordPress.org repository or verified commercial vendors. Nulled (pirated) plugins and themes must never be used.

Rationale: Nulled and pirated plugins are a leading vector for malware distribution. They frequently contain backdoors, cryptominers, SEO spam injectors, and other malicious code. Even legitimate-appearing free plugins from unofficial sources may be trojanized.

Audit:

This is a manual check. Review all installed plugins and themes:

```
$ wp plugin list --fields=name,status,version,update_available --path=/path/to/wordpress
```

Verify each plugin is available in the WordPress.org repository or from a known commercial vendor.

Remediation:

1. Audit all installed plugins and themes for their source. 2. Remove any plugins not traceable to a legitimate source. 3. Establish an approved plugin list for the organization. 4. Block plugin installation from the admin interface (see 4.1: DISALLOW_FILE_MODS).

Default Value: WordPress allows installation from any ZIP file by default.

1.13.0.3 8.3 Ensure plugin and theme updates are applied promptly Profile Applicability: Level 1

Assessment Status: Manual

Description: Security updates for plugins and themes should be applied within 72 hours of release. Critical security updates should be applied immediately or virtual-patched within 24 hours.

Rationale: Known vulnerabilities in popular WordPress plugins are actively exploited within hours of public disclosure. Delayed patching is the most common technical root cause of WordPress compromises.

Impact: Updates may occasionally introduce regressions or compatibility issues. Use a staging environment for testing when possible, but do not delay critical security patches.

Audit:

```
$ wp plugin list --fields=name,version,update_available --path=/path/to/wordpress
```

```
$ wp theme list --fields=name,version,update_available --path=/path/to/wordpress
```

Verify no security updates are pending.

Remediation:

1. Enable auto-updates for plugins and themes where supported. 2. Subscribe to vulnerability notification services (Patchstack, WPScan, Wordfence). Use the Exploit Prediction Scoring System (EPSS) alongside CVSS to prioritize remediation by real-world exploitability. 3. Establish a maintenance schedule for manual update review (weekly minimum). 4. Deploy virtual patching for critical vulnerabilities that cannot be patched immediately.

Default Value: Plugin and theme auto-updates are disabled by default (can be enabled per-plugin).

References: https://patchstack.com/database/

1.13.0.4 8.4 Ensure a Software Bill of Materials (SBOM) is maintained Profile Applicability: Level 2

Assessment Status: Manual

Description: A machine-readable Software Bill of Materials (SBOM) should be maintained for each WordPress deployment, documenting all components including WordPress core version, all active and inactive plugins and themes, third-party libraries bundled within plugins or themes, PHP version and extensions, and web server and database versions.

Rationale: Supply chain compromise accounted for 15% of breaches in IBM's Cost of a Data Breach Report (2025) with an average cost of $4.91 million, and third-party involvement doubled to 30% per the Verizon DBIR (2025). An SBOM enables rapid identification of affected components when a vulnerability is disclosed in any dependency, reducing the time to assess exposure and apply patches.

Impact: Requires tooling and process to generate and maintain the SBOM. Can be automated through `wp-cli` scripts and CI/CD pipeline integration.

Audit:

This is a manual check. Verify that: 1. An SBOM document or automated generation process exists for the WordPress deployment. 2. The SBOM is updated when plugins, themes, or core are added, updated, or removed. 3. The SBOM is stored in a location accessible to the security team.

Remediation:

1. Generate an SBOM using `wp-cli` and system commands:

```
# Core version
wp core version --path=/path/to/wordpress

# All plugins with versions
wp plugin list --fields=name,version,status --path=/path/to/wordpress --format=json

# All themes with versions
wp theme list --fields=name,version,status --path=/path/to/wordpress --format=json

# PHP version and extensions
php -v && php -m

# Web server and database versions
nginx -v 2>&1 || apache2 -v 2>&1
mysql --version
```

2. Integrate SBOM generation into deployment pipelines.
3. Cross-reference the SBOM against vulnerability databases (Patchstack, WPScan) on a regular schedule.

Default Value: No SBOM is maintained by default.

1.14 9 Web Application Firewall

This section addresses the deployment and configuration of a Web Application Firewall (WAF) to protect the WordPress application.

1.14.0.1 9.1 Ensure Web Application Firewall is Configured Profile Applicability: Level 2

Assessment Status: Manual

Description: Deploy a Web Application Firewall (WAF). This can be a server-level solution like **ModSecurity** with the OWASP Core Rule Set (CRS), or a cloud-based solution such as **Cloudflare WAF**, **Akamai**, or **Sucuri**.

Rationale: A WAF provides immediate protection against common web attacks (SQLi, XSS, RCE) before they reach the application. For server-level WAFs like ModSecurity, WordPress-specific exclusion rules are necessary to allow legitimate functionality (like post saving and media uploads) to pass through without being blocked as false positives. Cloud WAFs typically manage these rulesets automatically.

Impact: WAFs can introduce latency and false positives. Tuning is required. Cloud WAFs may require DNS changes.

Audit:

This is a manual check. Verify that: 1. A WAF is active and blocking malicious requests (verify via logs or simulation). 2. If using ModSecurity, the OWASP Core Rule Set and WordPress Rule Exclusions are enabled. 3. If using a Cloud WAF, the WordPress-specific protection profile is active.

Remediation:

For server-level WAF: 1. Install ModSecurity and the OWASP Core Rule Set. 2. Enable the WordPress exclusion rule set. - For OWASP CRS v3.x: Uncomment the WordPress exclusion rule in `crs-setup.conf`. - For OWASP CRS v4.x: Use the WordPress Rule Exclusions Plugin.

For Cloud WAF: 1. Route traffic through a provider such as Cloudflare, Akamai, or Sucuri. 2. Enable Managed Rulesets related to WordPress and OWASP Top 10.

Default Value: No WAF is configured by default.

Note: While this benchmark classifies WAF as Level 2, enterprise WordPress deployments should treat WAF as a baseline requirement. See the companion WordPress Security Architecture and Hardening Guide for extended enterprise guidance.

References: https://coreruleset.org/ https://github.com/coreruleset/wordpress-rule-exclusions-plugin

---

1.15 10 Backup and Recovery

This section addresses backup strategy, offsite storage, and recovery procedures for WordPress deployments.

1.15.0.1 10.1 Ensure backup and recovery procedures are implemented Profile Applicability: Level 1

Assessment Status: Manual

Description: Automated backups of the complete WordPress installation (files and database) must be performed regularly, stored offsite, encrypted, and tested for successful restoration on a recurring schedule.

Rationale: In the event of a security breach, the most reliable recovery strategy is to identify the root cause, verify the integrity of backups, and rebuild the compromised system from a known-good state. Without tested backups, recovery from ransomware, data corruption, or a complete site compromise may be impossible.

Impact: Backup processes consume storage and may briefly impact site performance during execution. Server-level backups are preferred over WordPress plugin-based backups for reliability and scope.

Audit:

This is a manual check. Verify that: 1. Automated backups are running on a defined schedule (daily minimum for most sites). 2. Backups include both the database and the full file system (WordPress core, `wp-content/`, and `wp-config.php`). 3. Backups are stored offsite, in a location inaccessible from the production server. 4. Backup data is encrypted both in transit and at rest. 5. Backup restoration has been tested within the last quarter. 6. Multiple backup generations are retained with sufficient history to recover from undetected compromises.

Remediation:

1. Configure server-level backups (not relying solely on WordPress plugins) with daily frequency at minimum.
2. Store backups in an offsite location (e.g., a separate cloud storage account, S3 bucket, or remote server) that is not accessible from the production web server.
3. Encrypt backup archives using AES-256 or equivalent.
4. Test backup restoration quarterly by performing a full recovery to a staging environment.
5. Retain at least 30 days of daily backups and 90 days of weekly backups, allowing recovery from compromises that may not be detected immediately.
6. Document the recovery procedure and assign clear ownership and responsibilities.

Default Value: No backups are configured by default. Backup responsibility depends on the hosting environment.

1.16 11 AI and Generative AI Security

AI tools are increasingly integrated into WordPress workflows for content generation, chatbots, code assistance, and site management. IBM's Cost of a Data Breach Report (2025) found that 13% of organizations experienced a breach involving an AI model or application, and 97% of those breaches involved systems lacking proper access controls. This section provides controls for securing AI integrations in WordPress environments.

1.16.0.1 11.1 Ensure AI API keys are securely stored Profile Applicability: Level 1

Assessment Status: Automated

Description: API keys and credentials for AI/LLM services (OpenAI, Anthropic, Google, etc.) must be stored securely and never exposed in client-side code, version control, or the WordPress database.

Rationale: AI API keys grant access to paid services and may allow data exfiltration or abuse. The Verizon DBIR (2025) found that leaked secrets in code repositories had a median remediation time of 94 days, with 66% being JSON Web Tokens. AI API keys are similarly at risk.

Impact: Requires using `wp-config.php` constants or environment variables rather than storing keys in plugin settings (database).

Audit:

1. Search the codebase and database for AI service API keys:

```
grep -r "sk-" /path/to/wordpress/wp-content/ --include="*.php"
wp db query "SELECT option_name, option_value FROM wp_options WHERE option_value LIKE '%sk-%'
```

2. Verify API keys are defined as constants in `wp-config.php` or loaded from environment variables.
3. Confirm `.gitignore` excludes `wp-config.php` and environment files.

Remediation:

1. Move all AI API keys to `wp-config.php` constants (e.g., `define('OPENAI_API_KEY', getenv('OPENAI_API_KEY'));`).
2. Remove any API keys stored in the `wp_options` table or in plugin settings screens.
3. Rotate any keys that have been exposed in version control or client-side code.

Default Value: Most AI plugins store API keys in the database via the WordPress settings API.

1.16.0.2 11.2 Ensure AI-generated content is sanitized Profile Applicability: Level 1

Assessment Status: Manual

Description: All content generated by AI services must be treated as untrusted input and sanitized before rendering or storage, using WordPress's standard escaping functions (`esc_html()`, `wp_kses()`, `esc_attr()`, etc.).

Rationale: AI models can generate content that includes XSS payloads, SQL injection patterns, or malicious HTML — either through prompt injection attacks or as a natural consequence of model behavior. Treating AI output as trusted input bypasses WordPress's defense-in-depth sanitization model.

Impact: Minimal. Requires using existing WordPress sanitization functions on AI output, which is consistent with standard development practice for any external data source.

Audit:

Review custom AI integration code for direct output of AI-generated content without sanitization. Check that AI responses pass through WordPress escaping functions before being stored in post content, meta fields, or rendered in templates.

Remediation:

1. Apply `wp_kses_post()` to AI-generated content before storing in `post_content`.
2. Apply `esc_html()` or `esc_attr()` before rendering in HTML templates.
3. Never pass AI-generated content directly to `$wpdb->query()` without `$wpdb->prepare()`.

Default Value: No WordPress-specific defaults; depends on plugin implementation.

1.16.0.3 11.3 Ensure AI tool usage is governed by policy Profile Applicability: Level 2

Assessment Status: Manual

Description: Organizations must maintain and enforce a policy governing the use of AI tools by WordPress team members and integrated into WordPress sites. The policy must address approved tools, data classification for AI inputs, authentication requirements, and disclosure of AI-generated content.

Rationale: Shadow AI — the unsanctioned use of AI tools by employees — is a measurable risk. IBM's Cost of a Data Breach Report (2025) found that 20% of breached organizations experienced a shadow AI incident, adding $200,000 to average breach costs ($670,000 for organizations with high shadow AI prevalence), and that 63% of organizations lack AI governance policies. The Verizon DBIR (2025) found that 15% of employees routinely access GenAI systems on corporate devices, with 72% using non-corporate email accounts.

Impact: Requires organizational policy development and enforcement. May restrict which AI plugins can be installed on WordPress sites.

Audit:

1. Verify that a written AI acceptable use policy exists and has been communicated to all WordPress team members.
2. Confirm that only approved AI plugins are installed on production sites.
3. Verify that AI service authentication uses corporate SSO or managed API keys (not personal accounts).

Remediation:

1. Develop an AI acceptable use policy covering approved tools, prohibited data inputs (credentials, PII, proprietary content), and disclosure requirements.
2. Maintain an inventory of approved AI plugins and services.
3. Configure AI services with corporate authentication and audit logging where available.
4. Include AI governance in security awareness training.

Default Value: No AI governance policy exists by default.

1.17 12 Server Access and Network

This section addresses secure remote access to the server hosting WordPress and host-level network controls.

1.17.0.1 12.1 Ensure SSH key-based authentication is enforced Profile Applicability: Level 1

Assessment Status: Automated

Description: SSH access to the server must use key-based authentication only. Password-based SSH authentication must be disabled.

Rationale: Password-based SSH authentication is vulnerable to brute-force and credential stuffing attacks. Key-based authentication is significantly more resistant to these attacks and is the standard for secure server access.

Impact: All administrators must generate and deploy SSH key pairs before password authentication is disabled. Emergency access procedures should be documented.

Audit:

```
$ grep -E 'PasswordAuthentication|PubkeyAuthentication' /etc/ssh/sshd_config
```

Verify: `PasswordAuthentication no` and `PubkeyAuthentication yes`.

Remediation:

In `/etc/ssh/sshd_config`:

```
PasswordAuthentication no
PubkeyAuthentication yes
PermitRootLogin no
```

Restart the SSH service:

```
$ sudo systemctl restart sshd
```

Default Value: Password authentication is enabled by default on most Linux distributions.

1.17.0.2 12.2 Ensure SFTP is used and FTP is disabled Profile Applicability: Level 1

Assessment Status: Automated

Description: File transfers to and from the server must use SFTP (SSH File Transfer Protocol) or SCP. Plain FTP and FTPS must be disabled. No FTP server software should be installed.

Rationale: FTP transmits credentials and data in cleartext, making it trivial to intercept on a network. Even FTPS (FTP over TLS) has known weaknesses in channel binding. SFTP operates over the encrypted SSH channel and requires no additional server software.

Impact: Users and deployment tools relying on FTP must be migrated to SFTP. Most modern FTP clients support SFTP natively.

Audit:

```
$ ss -tlnp | grep -E ':21\b'
```

Verify no service is listening on port 21.

```
$ dpkg -l | grep -iE 'vsftpd|proftpd|pure-ftpd'
```

Verify no FTP server packages are installed.

Remediation:

1. Remove any installed FTP server software:

```
$ sudo apt purge vsftpd proftpd-basic pure-ftpd 2>/dev/null
```

2. Verify SFTP is available through the SSH server (enabled by default in OpenSSH):

```
$ grep 'Subsystem.*sftp' /etc/ssh/sshd_config
```

3. Configure deployment pipelines and team workflows to use SFTP or SCP exclusively.

Default Value: FTP is not installed by default on most modern Linux distributions, but may be present in legacy environments.

1.17.0.3 12.3 Ensure a host-based firewall is configured Profile Applicability: Level 1

Assessment Status: Automated

Description: A host-based firewall (e.g., UFW on Ubuntu/Debian, firewalld on RHEL/CentOS) must be enabled and configured to restrict inbound traffic to only required ports: HTTP (80), HTTPS (443), and SSH on a non-standard port.

Rationale: A host-based firewall provides a final layer of network defense, blocking unauthorized access to services running on the server even if upstream network controls fail. Restricting SSH to a non-standard port reduces automated scanning noise.

Impact: Moving SSH to a non-standard port requires updating all SSH client configurations and deployment scripts. Misconfigured rules may lock out administrators.

Audit:

For UFW:

```
$ sudo ufw status verbose
```

Verify the firewall is active and only the required ports are open.

Remediation:

For UFW on Ubuntu/Debian:

```
# Enable UFW
sudo ufw default deny incoming
sudo ufw default allow outgoing

# Allow required ports
sudo ufw allow 443/tcp
sudo ufw allow 80/tcp
sudo ufw allow <custom-ssh-port>/tcp

# Enable the firewall
sudo ufw enable
```

Default Value: UFW is installed but inactive on Ubuntu. No firewall is configured by default on most distributions.

References: https://help.ubuntu.com/community/UFW

1.17.0.4 12.4 Ensure per-site process isolation is configured Profile Applicability: Level 2

Assessment Status: Manual

Description: Each WordPress site on the server should run under a dedicated system user account with its own PHP-FPM pool. Sites must not share the same PHP process user.

Rationale: Process isolation limits the blast radius of a compromise. If one site is breached, the attacker cannot read files, access databases, or modify code belonging to other sites on the same server. This is especially critical in multi-tenant hosting environments.

Impact: Requires per-site PHP-FPM pool configuration and dedicated system user accounts. Increases server resource usage proportionally to the number of sites.

Audit:

```
$ ps aux | grep php-fpm | grep -v grep
```

Verify each site's PHP-FPM pool runs as a different system user.

```
$ grep -r '^user' /etc/php/*/fpm/pool.d/
```

Verify each pool configuration specifies a unique user.

Remediation:

1. Create a dedicated system user for each site:

```
$ sudo useradd -r -s /usr/sbin/nologin site1_user
```

2. Configure a dedicated PHP-FPM pool for each site in `/etc/php/8.3/fpm/pool.d/site1.conf`:

```
[site1]
user = site1_user
group = site1_user
listen = /run/php/php8.3-fpm-site1.sock
listen.owner = www-data
listen.group = www-data
```

3. Set file ownership accordingly and restart PHP-FPM.

Default Value: A single `www-data` pool serves all sites by default.

1.18 13 Multisite Security

This section addresses security considerations specific to WordPress Multisite installations.

1.18.0.1 13.1 Ensure Super Admin accounts are minimized and audited Profile Applicability: Level 1

Assessment Status: Manual

Description: In a WordPress Multisite network, the number of Super Admin accounts must be limited to the absolute minimum required. Super Admin grants unrestricted access across all sites in the network, including the ability to install plugins and themes network-wide, create and delete sites, and manage all users across all sites.

Rationale: Compromising any single Super Admin account grants an attacker full control over every site in the Multisite network. Unlike single-site Administrator accounts, Super Admin access cannot be scoped or restricted — it is all-or-nothing across the entire network.

Impact: Operations that require Super Admin access must be performed by a small, audited group. Day-to-day site administration should use per-site Administrator roles.

Audit:

```
$ wp super-admin list --path=/path/to/wordpress
```

Review the list. Verify that each Super Admin account is actively needed and assigned to a specific individual.

Remediation:

1. Audit existing Super Admin accounts.
2. Remove Super Admin privileges from accounts that don't require network-level access:

```
$ wp super-admin remove <username> --path=/path/to/wordpress
```

3. Assign per-site Administrator roles for day-to-day operations.
4. Review Super Admin accounts quarterly.

Default Value: One Super Admin account is created during Multisite installation.

1.18.0.2 13.2 Ensure network-activated plugins are reviewed for cross-site impact Profile Applicability: Level 2

Assessment Status: Manual

Description: Plugins activated at the network level in a WordPress Multisite installation run on all sites in the network. Each network-activated plugin should be reviewed for its security impact across all sites, and network activation should be reserved for plugins that genuinely require network-wide operation.

Rationale: A vulnerability in a network-activated plugin affects every site in the Multisite network simultaneously. Shared database tables and a common file system mean that a single exploit can traverse all sites. Limiting network activation reduces the shared attack surface.

Impact: Plugins that are only needed on specific sites must be activated per-site rather than network-wide. This may require changes to existing deployment workflows.

Audit:

This is a manual check. Review the list of network-activated plugins:

```
$ wp plugin list --status=active-network --fields=name,version --path=/path/to/wordpress
```

For each network-activated plugin, verify that network-wide activation is necessary rather than per-site activation.

Remediation:

1. Review all network-activated plugins.
2. Deactivate plugins at the network level that don't require network-wide operation.
3. Activate them per-site only where needed.
4. Ensure domain mapping (if used) enforces TLS across all mapped domains.

Default Value: No plugins are network-activated by default.

1.19 Appendix A: Recommendation Summary

The following table summarizes all recommendations in this benchmark.

ID	Recommendation	Level	Assessment
1.1	Ensure TLS 1.2+ is enforced	L1	Automated
1.2	Ensure HTTP security headers are configured	L1	Automated
1.3	Ensure server tokens are hidden	L1	Automated
1.4	Ensure PHP execution is blocked in uploads	L1	Automated
1.5	Ensure rate limiting is configured for all APIs	L1	Automated
2.1	Ensure expose_php is disabled	L1	Automated
2.2	Ensure display_errors is disabled	L1	Automated
2.3	Ensure dangerous PHP functions are disabled	L1	Automated
2.4	Ensure open_basedir restricts file access	L2	Automated
2.5	Ensure PHP session security is configured	L1	Automated
3.1	Ensure DB user has minimal privileges	L1	Automated
3.2	Ensure DB is not externally accessible	L1	Automated
3.3	Ensure non-default table prefix is used	L1	Manual
3.4	Ensure database query logging is enabled	L2	Automated
4.1	Ensure DISALLOW_FILE_MODS is true	L1	Automated
4.2	Ensure FORCE_SSL_ADMIN is true	L1	Automated
4.3	Ensure debug mode is disabled	L1	Automated
4.4	Ensure XML-RPC is disabled	L1	Automated
4.5	Ensure automatic core updates are enabled	L1	Automated
4.6	Ensure unique auth keys and salts are configured	L1	Automated
4.7	Ensure wp-cron.php is replaced with system cron	L1	Automated
5.1	Ensure 2FA is required for administrators	L1	Manual
5.2	Ensure admin accounts are minimized	L1	Manual
5.3	Ensure max session lifetime is enforced	L1	Automated
5.4	Ensure user enumeration is prevented	L1	Automated
5.5	Ensure reauthentication for privileged actions	L2	Manual
5.6	Ensure unauthenticated REST API is restricted	L2	Automated

ID	Recommendation	Level	Assessment
5.7	Ensure strong password policy is enforced	L1	Manual
5.8	Ensure roles and capabilities are defined in code	L2	Manual
6.1	Ensure files are owned by non-web-server user	L1	Automated
6.2	Ensure `wp-config.php` has restrictive permissions	L1	Automated
6.3	Ensure `wp-config.php` is above document root	L2	Manual
7.1	Ensure user activity logging is enabled	L1	Manual
7.2	Ensure file integrity monitoring is configured	L2	Automated
7.3	Ensure server-level malware detection is configured	L2	Manual
8.1	Ensure unused plugins and themes are removed	L1	Automated
8.2	Ensure extensions are from trusted sources	L1	Manual
8.3	Ensure plugin/theme updates are applied promptly	L1	Manual
8.4	Ensure a Software Bill of Materials is maintained	L2	Manual
9.1	Ensure Web Application Firewall is configured	L2	Manual
10.1	Ensure backup and recovery procedures are implemented	L1	Manual
11.1	Ensure AI API keys are securely stored	L1	Automated
11.2	Ensure AI-generated content is sanitized	L1	Manual
11.3	Ensure AI tool usage is governed by policy	L2	Manual
12.1	Ensure SSH key-based authentication is enforced	L1	Automated
12.2	Ensure SFTP is used and FTP is disabled	L1	Automated
12.3	Ensure a host-based firewall is configured	L1	Automated
12.4	Ensure per-site process isolation is configured	L2	Manual
13.1	Ensure Multisite Super Admin accounts are minimized	L1	Manual
13.2	Ensure network-activated plugins are reviewed	L2	Manual

1.20 Related Documents

- **WordPress Security Architecture and Hardening Guide** — Enterprise-focused security architecture and hardening guide covering threat landscape, OWASP Top 10 coverage, server hardening, authentication, supply chain, incident response, and AI security. This benchmark's technical controls complement the Hardening Guide's broader strategic guidance.
- **WordPress Security Style Guide** — Principles, terminology, and formatting conventions for writing about WordPress security.
- **WordPress Security White Paper (WordPress.org, September 2025)** — The official upstream document describing WordPress core security architecture, maintained at developer.wordpress.org.

1.21 License and Attribution

Sources and Acknowledgments: The format of this benchmark is adapted from industry-standard benchmarks published by the Center for Internet Security (CIS). Technical guidance draws on the OWASP Top 10 (2025), NIST SP 800-63B, the Verizon Data Breach Investigations Report (2025), and IBM's Cost of a Data Breach Report (2025).