SOURCE CODE REVIEW & REGEX GUIDE

Topics Covered:

✦ Introduction & Methodology	✦ Regex Fundamentals
✦ Critical Vulnerability Patterns	✦ Regex for Security Testing
✦ Language-Specific Review Tips	✦ Regex Cheat Sheet (All Languages)
✦ Common Vulnerabilities & Key Areas	✦ Reporting & Documentation
✦ Black-Box vs White-Box Analysis	✦ Code Review Signatures & Patterns

Author

Abdalla Abdelrhman

Cybersecurity Consultant

linkedin.com/in/0x2nac0nda

Published: February 2026

01 INTRODUCTION — What is Source Code Review?

Source Code Review is one of the most powerful and cost-effective security practices in software development. Unlike penetration testing that targets running applications, source code review analyzes the actual code to find vulnerabilities before deployment — making it a proactive and essential security layer.

In various situations it may be possible to perform a source code audit to help attack a target web application: some applications are open source, enabling you to download their code and scour it for vulnerabilities. During a penetration test, the application owner may grant access to source code to maximize audit effectiveness. You may also discover a file disclosure vulnerability that enables downloading source code, or review client-side code such as JavaScript which is accessible without privileged access.

Why Should You Learn Source Code Review?

- Proactive security — finds bugs before they reach production
- Cost-effective — fixing bugs early is 100x cheaper than after deployment
- Interview advantage — top cybersecurity companies test this skill
- Works on all languages — the methodology is universal
- Combines with Regex — automates pattern-based vulnerability detection
- Complements black-box testing — reveals issues extremely difficult to detect from the outside

💡 Key Insight

It is often believed that to carry out a code review, you must be an experienced programmer. However, many higher-level languages can be read by someone with limited programming experience. Many vulnerability types manifest themselves in the same way across all languages. A cheat sheet can help understand language-specific syntax and APIs.

02 BLACK-BOX vs WHITE-BOX TESTING

Understanding the difference between black-box and white-box testing is essential for effective security assessment. Each approach has inherent strengths, and combining both yields the best results.

▸ Black-Box Testing

Black-box testing involves attacking the application from the outside and monitoring its inputs and outputs, with no prior knowledge of its inner workings. Using automated fuzzing techniques, it is possible to send an application hundreds of test cases per minute, which propagate through all relevant code paths and return a response immediately. By sending triggers for common vulnerabilities to every field in every form, it is often possible to find within minutes a mass of problems that would take days to uncover via code review.

▸ White-Box Testing (Code Review)

A white-box approach involves looking inside the application's internals, with full access to design documentation, source code, and other materials. With access to source code, it is often possible to quickly locate problems that would be extremely difficult or time-consuming to detect using only black-box techniques. For example, a backdoor password that grants access to any user account may be easy to identify by reading the code but nearly impossible to detect using a password-guessing attack.

▸ Why White-Box Is Not a Full Substitute

Many enterprise-class applications have a complex structure with numerous layers of processing of user-supplied input. Different controls and checks are implemented at each layer, and what appears to be a clear vulnerability in one piece of source code may be fully mitigated by code elsewhere. Therefore, code review is usually not an effective substitute for black-box testing on its own.

▸ The Combined Approach

In most situations, black-box and white-box techniques complement and enhance each other. Having found a prima facie vulnerability through code review, the easiest way to establish whether it is real is to test for it on the running application. Conversely, having identified some anomalous behavior on a running application, the easiest way to investigate its root cause is to review the relevant source code.

🎯 Best Practice

Allow the time and effort you devote to each approach to be guided by the application's behavior during hands-on testing, and the size and complexity of the codebase. Aim to combine a suitable mix of black- and white-box techniques for maximum effectiveness.

03 SOURCE CODE REVIEW METHODOLOGY

A key objective of effective code review is to identify as many security vulnerabilities as possible, given a certain amount of time and effort. You must take a structured approach, using various techniques to ensure that the 'low-hanging fruit' within the codebase is quickly identified, leaving time for issues that are more subtle and harder to detect.

A threefold approach to auditing a web application codebase is effective in identifying vulnerabilities quickly: (1) Tracing user-controllable data from its entry points; (2) Searching for signatures indicating common vulnerabilities; (3) Performing line-by-line review of inherently risky code.

▸ Phase 1: Reconnaissance & Preparation

- Understand the application purpose, architecture, and tech stack
- Map data flow from user input → processing → output → database
- Identify critical components: authentication, payments, admin panels
- Review previous security findings and known vulnerabilities
- Establish the extent of any customization — library classes, wrappers, custom mechanisms for session handling

▸ Phase 2: Automated Scanning

- Run SAST tools: Semgrep, SonarQube, Bandit, ESLint security plugin
- Scan dependencies: OWASP Dependency Check, Snyk, npm audit
- Search for hardcoded secrets: Trufflehog, GitLeaks, YARA rules
- Use Regex patterns to search for signatures of common vulnerabilities

▸ Phase 3: Manual Code Review

- Trace user-controllable data from entry points into the application and review processing code
- Review input validation and sanitization logic at every layer
- Analyze authentication and session management — look for custom mechanisms
- Check authorization and access control enforcement on every endpoint
- Verify cryptographic implementations and key storage
- Review error handling to ensure no sensitive data leaks in responses
- Check output encoding to prevent XSS in returned HTML

▸ Phase 4: Vulnerability Classification

- Map findings to OWASP Top 10 and CWE categories
- Assign severity: Critical → High → Medium → Low → Info
- Calculate business impact for each vulnerability
- Prioritize by exploitability and potential damage

▸ Phase 5: Reporting & Remediation

- Document file path, line number, and vulnerable code snippet

- Explain the vulnerability with a technical description
- Provide proof-of-concept exploitation scenario
- Include secure code fix and remediation guidance

04 CRITICAL VULNERABILITY PATTERNS

Many types of web application vulnerabilities have a fairly consistent signature within the codebase. This means you can normally identify a good portion of an application's vulnerabilities by quickly scanning and searching. The examples here are language-neutral — what matters is the programming technique being employed.

▸ 1. Hardcoded Secrets & Credentials

- API keys, passwords, tokens directly in source code
- Database connection strings with embedded credentials
- Private keys or certificates stored in code files
- Backdoor passwords used for testing or administrative purposes — these usually stand out when you review credential validation logic
- JWT secret keys or session encryption keys

```
// ❌ VULNERABLE — Backdoor password in validation logic
private UserProfile validateUser(String username, String password) {
    UserProfile up = getUserProfile(username);
    if (checkCredentials(up, password) || "oculiomnium".equals(password))
        return up;   // Backdoor: ANY user authenticated with this password!
    return null;
}

// ✅ SECURE — No backdoor, proper credential check only
private UserProfile validateUser(String username, String password) {
    UserProfile up = getUserProfile(username);
    if (checkCredentials(up, password)) return up;
    return null;
}
```

▸ 2. Cross-Site Scripting (XSS)

In the most obvious examples of XSS, parts of the HTML returned to the user are explicitly constructed from user-controllable data. In more subtle cases, user-controllable data is used to set a variable that is later used to build the response. XSS flaws can be conditional — triggered only when another parameter has a specific value — making them very hard to find via standard fuzz testing.

```
// ❌ VULNERABLE — User data directly in HTML construction
String link = "<a href=" + HttpUtility.UrlDecode(
    Request.QueryString["refURL"]) + "&SiteID=" + SiteId + "</a>";
objCell.InnerHtml = link;  // XSS: refURL injected into HTML

// ❌ SUBTLE XSS — Conditional trigger (only when type=3)
if ("3".equals(requestType) && request.getParameter("title") != null)
    m_pageTitle = request.getParameter("title");  // Used later in page
```

▸ 3. SQL Injection

SQL injection vulnerabilities most commonly arise when hard-coded strings are concatenated with user-controllable data to form a SQL query. A simple way to identify this is to search for hard-coded SQL substrings such as: "SELECT , "INSERT , "DELETE , " AND , " OR , " WHERE , " ORDER BY — appending a space reduces false positives. Searches should be case-insensitive since SQL keywords are processed case-insensitively.

```
// ❌ VULNERABLE — String concatenation in SQL query
StringBuilder SqlQuery = new StringBuilder(
    "SELECT name, accno FROM TblCustomers WHERE " + SqlWhere);
if (Request.QueryString["CID"] != null) {
    SqlQuery.Append(" AND CustomerID = ");
    SqlQuery.Append(Request.QueryString["CID"].ToString());
}

// ✅ SECURE — Parameterized query
PreparedStatement ps = conn.prepareStatement(
    "SELECT name, accno FROM TblCustomers WHERE CustomerID = ?");
ps.setString(1, customerId);
```

▸ 4. Path Traversal

The usual signature involves user-controllable input being passed to a filesystem API without validation. User data is appended to a hard-coded directory path, enabling an attacker to use dot-dot-slash sequences to step up the directory tree. Review any functionality that enables users to upload or download files, and search for all file APIs in the relevant language.

```
// ❌ VULNERABLE — User input directly in file path
FileStream fs = new FileStream(SpreadsheetPath +
    HttpUtility.UrlDecode(Request.QueryString["AttachName"]),
    FileMode.Open, FileAccess.Read);
// Attack: AttachName = "../../../../../../etc/passwd"

// ✅ SECURE — Validate and sanitize filename
string name = Path.GetFileName(Request.QueryString["AttachName"]);
string fullPath = Path.Combine(SpreadsheetPath, name);
if (!fullPath.StartsWith(SpreadsheetPath)) throw new Exception();
```

▸ 5. Arbitrary Redirection

Phishing vectors such as arbitrary redirects are often easy to spot in source code. User-supplied data from the query string is used to construct a URL to which the user is redirected. Watch for canonicalization performed after validation — this often leads to bypass. Often, redirects can be found by inspecting client-side JavaScript without any special access.

```
// ❌ VULNERABLE — User-controlled redirect URL
httpResponse.Redirect(HttpUtility.UrlDecode(
    Request.QueryString["refURL"]) + "&SiteCode=" + ...);
```

```
// ❌ FLAWED FIX — Canonicalization AFTER validation allows bypass
// Script checks for '//' but then calls unescape() again after!
// Attack payload: ?redir=http:%25252f%25252fattacker.com
```

▸ 6. OS Command Injection

Code that interfaces with external systems often contains signatures indicating code injection flaws. Search for system(), exec(), popen(), shell_exec() and similar APIs. If user-controllable data is passed unfiltered, the application is vulnerable to arbitrary command execution. Commands can be chained using the | character.

```
// ❌ VULNERABLE — User input in system command
void send_mail(const char *message, const char *addr) {
    char cmd[4096];
    snprintf(cmd, 4096, "echo '%s' | sendmail %s", message, addr);
    system(cmd);  // Command injection via addr parameter
}

// ✅ SECURE — Use safe API with argument arrays
subprocess.run(['sendmail', addr], input=message, shell=False)
```

▸ 7. Missing Input Validation

- No validation before processing user input at any layer
- Missing length limits on text fields
- No type checking or sanitization
- Trusting client-side validation only — server must enforce independently

▸ 8. Insecure Cryptography

- Using weak algorithms: MD5, SHA1, DES — search for md5(), sha1() calls
- Hardcoded encryption keys or initialization vectors (IV)
- Using ECB mode instead of CBC or GCM
- Weak random number generators — e.g., java.util.Random is predictable and NOT cryptographically secure

▸ 9. Missing Authorization Checks

- Admin functions accessible without role verification
- Missing object-level authorization (IDOR vulnerability)
- No server-side access control enforcement

▸ 10. Source Code Comments Revealing Vulnerabilities

Many software vulnerabilities are actually documented within source code comments. Developers may record a reminder to fix a problem later but never get around to it. Search for: bug, problem, bad, hope, todo, fix, overflow, crash, inject, xss, trust.

```
// ❌ VULNERABLE — Developer left a warning comment!
char buf[200]; // I hope this is big enough
```

```
...
strcpy(buf, userinput);  // Buffer overflow if userinput > 200 bytes!

// Search terms to find similar issues in any codebase:
// bug | problem | bad | hope | todo | fix | overflow | crash | inject | xss |
trust
```

▸ 11. Native Software Bugs (C/C++)

Buffer overflow vulnerabilities typically employ unchecked APIs: strcpy, strcat, memcpy, sprintf and their variants. Search for all uses and verify whether the source buffer is user-controllable and whether the destination buffer is large enough. Note: even 'safe' alternatives like strncpy can be used incorrectly — a thorough audit requires line-by-line review tracing every operation on user data.

```
// ❌ VULNERABLE — No bounds check on user input
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName) {
    char strFileName[MAX_PATH];
    strcpy(strFileName, pszName);  // Overflow if pszName > MAX_PATH!
}

// ❌ STILL VULNERABLE — strncpy misused (uses strlen, not sizeof)
    strncpy(strFileName, pszName, strlen(pszName)); // Wrong length!

// ❌ INTEGER BUG — signed vs unsigned comparison
    if (len < sizeof(strFileName))  // len is signed int!
        strcpy(strFileName, pszName); // Negative len passes check!
```

05 COMMON VULNERABILITIES — All Languages

These vulnerabilities appear across all programming languages. Understanding them is essential for any source code review.

▸ Common Vulnerabilities by Category

Vulnerability	Description & Impact
SQL Injection	Unsanitized input in SQL queries allows database manipulation, data theft, or destruction
Cross-Site Scripting (XSS)	Malicious scripts injected into web pages steal cookies, hijack sessions, or deface content
CSRF	Attacker tricks authenticated user into performing unintended actions using their session
SSRF	Server makes requests to internal/external URLs controlled by attacker, exposing internal services
Path Traversal	Accessing files outside intended directory using ../ sequences to read sensitive data
Command Injection	OS commands injected through user input via system(), exec(), or shell functions
Insecure Deserialization	Malicious serialized data exploits deserialization logic for RCE or data manipulation
XXE Injection	XML External Entity injection reads local files or performs SSRF through XML parsers
IDOR / BOLA	Accessing other users' resources by manipulating object IDs without authorization checks
Broken Access Control	Missing or improper authorization allows access to restricted functions or data
Mass Assignment	Extra parameters in requests modify unintended fields like role=admin or isAdmin=true
Information Disclosure	Verbose errors, debug info, or stack traces leak sensitive system information
Insecure File Upload	Uploading executable files (PHP, JSP) that run on server leads to Remote Code Execution
Weak Cryptography	Using MD5, SHA1, DES or hardcoded keys makes encrypted data vulnerable to decryption
Arbitrary Redirection	User-controlled URL in redirect response enables phishing attacks against application users

▸ Key Areas to Review Per Vulnerability

Vulnerability	Where to Look	What to Check
SQL Injection	`Database query files, ORM configs`	String concat vs parameterized queries
XSS	`Template files, output rendering`	Output encoding, CSP headers, sanitization
CSRF	`Form handlers, state-changing endpoints`	CSRF tokens, SameSite cookies, Origin checks
SSRF	`URL fetch functions, API calls`	URL validation, whitelist, internal access
Path Traversal	`File operations, download handlers`	Input sanitization, path canonicalization
Command Injection	`System calls, shell execution`	Input validation, avoid shell=true
Auth Bypass	`Login, token validation, session`	Token expiry, signature validation, MFA
IDOR	`Object access, API endpoints`	Server-side ownership verification
File Upload	`Upload handlers, storage config`	File type validation, rename, exec prevention
Redirection	`Redirect APIs, Location headers`	URL validation, whitelist of allowed domains

06 CRITICAL SECURITY PATTERNS TO LOOK FOR

During any source code review, these three categories of security patterns must be thoroughly checked. Missing any of these is a critical oversight.

▸ Hardcoded Secrets

- API keys, passwords, tokens directly written in source code files
- Database connection strings with embedded username and password
- Private keys, certificates, or PEM files referenced in code
- JWT secret keys or session encryption keys hardcoded
- Cloud provider credentials (AWS access keys, Azure, GCP keys)
- Third-party service tokens (Stripe, SendGrid, Twilio)
- Internal service URLs with embedded authentication

```
// ❌ VULNERABLE — Secrets exposed in code
const stripe = require('stripe')('sk_live_abc123secret');
const AWS_KEY = 'AKIAIOSFODNN7EXAMPLE';
const DB = 'mongodb://admin:password123@db.server.com';

// ✅ SECURE — Secrets from environment
const stripe = require('stripe')(process.env.STRIPE_KEY);
const AWS_KEY = process.env.AWS_ACCESS_KEY_ID;
const DB = process.env.MONGODB_URI;
```

▸ Insecure Coding Patterns

- String concatenation used to build SQL queries with user input
- Using eval(), exec(), or Function() with dynamic input
- Disabling SSL/TLS certificate verification (verify=false)
- Trusting only client-side validation without server-side checks
- Missing or overly verbose error handling exposing stack traces
- Logging sensitive data: passwords, tokens, PII, credit cards
- Using deprecated or insecure cryptographic algorithms (MD5, SHA1, DES)
- Deserializing untrusted data without validation (pickle, Marshal, ObjectInputStream)
- Using shell=true or equivalent when executing system commands
- Storing passwords in plain text or using reversible encoding (Base64)

```
// ❌ INSECURE PATTERNS
eval(userInput);                          // Code injection
os.system('ping ' + userInput);          // Command injection
requests.get(url, verify=False);         // SSL disabled
console.log('Password:', password);      // Logging secrets
hash = md5(password);                    // Weak hash
data = pickle.loads(untrustedBytes);     // Unsafe deserialization
```

```
// ✅ SECURE PATTERNS
JSON.parse(userInput);                     // Safe parsing only
subprocess.run(['ping', userInput]);       // No shell injection
requests.get(url, verify=True);            // SSL enabled
logger.info('Login attempt');              // No sensitive data
hash = bcrypt.hash(password, 12);          // Strong hash
data = safe_deserialize(trustedBytes);     // Validated input
```

▸ Missing Security Controls

- No input validation or sanitization before processing user data
- Missing authorization checks on admin or sensitive endpoints
- No rate limiting on authentication or sensitive API operations
- Missing CORS restrictions allowing any origin to access API
- No output encoding before rendering data in HTML responses
- Disabled or missing security headers (HSTS, CSP, X-Frame-Options)
- No secure flag on session or authentication cookies
- Missing token expiration or refresh mechanisms
- No audit logging for security-critical operations
- File uploads stored in web-accessible directory without restrictions

```
// ❌ MISSING SECURITY CONTROLS
app.get('/admin/users', (req, res) => {
    // No authorization check — anyone can access!
    res.json(getAllUsers());
});

// ✅ PROPER SECURITY CONTROLS
app.get('/admin/users', authMiddleware, roleCheck('admin'),
    rateLimiter, (req, res) => {
    auditLog('admin_access', req.user.id);
    res.json(getAllUsers());
});
```

07 LANGUAGE-SPECIFIC CODE REVIEW TIPS

Each language has specific APIs and patterns that are security-critical. Applications may extend library classes, implement wrappers to standard API calls, and implement custom security mechanisms. Establish the extent of such customization before diving into detail.

Python (Django/Flask)	eval()/exec() usage, pickle deserialization, raw SQL in Django ORM, Jinja2 SSTI, os.system() calls
JavaScript (Node.js)	child_process injection, NoSQL injection in MongoDB, prototype pollution, eval() usage, JWT secrets
Java (Spring)	Deserialization vulnerabilities, XML parser (XXE), LDAP injection, Spring Security misconfigs, Runtime.exec()
PHP (Laravel)	File inclusion (require/include with user input), SQL injection, session handling, eval(), register_globals, magic_quotes
C# (.NET)	Deserialization in ASP.NET, raw SQL in ADO.NET, missing HTTPS enforcement, Process.Start() with user input
Go (Golang)	Race conditions in goroutines, SQL injection, insecure HTTP config, path traversal in file operations
Ruby (Rails)	Mass assignment, Marshal deserialization, ActiveRecord SQL injection, CSRF bypass
Swift (iOS)	Insecure Keychain usage, hardcoded credentials, missing ATS, sensitive data in logs
Kotlin (Android)	Exported components, SharedPreferences exposure, WebView injection, missing certificate pinning
C / C++	Buffer overflow (strcpy, strcat, memcpy), use-after-free, integer overflow, unsafe functions, format string bugs
Perl	open() with pipe in filename enables command injection, taint mode bypass via weak regex, eval() with user input

08 DANGEROUS APIs BY PLATFORM

Every platform has APIs that can introduce security vulnerabilities if used unsafely. This section maps the most dangerous APIs across the major web development platforms.

▸ Java — User Input Sources

API	Description
getParameter / getParameterValues	URL query string and POST body parameters — primary user input source
getQueryString	Returns the entire raw query string
getHeader / getHeaders	HTTP headers — can contain attacker-controlled data
getCookies	Cookie values — attacker-controllable, review for injection
getRequestURI / getRequestURL	Full URL including query string
getInputStream / getReader	Raw request body — access to all submitted data
getRemoteUser / getUserPrincipal	If users choose their own username, this can introduce malicious input

▸ Java — Dangerous APIs

API / Class	Risk & Description
Statement.execute / executeQuery	SQL injection if user input is concatenated into query string
Connection.prepareStatement	SAFE alternative — use parameter placeholders (?) with setString()
Runtime.getRuntime().exec()	OS command execution — vulnerable if user controls the command string
java.io.File constructor	Path traversal if user data passed as filename without validation
HttpServletResponse.sendRedirect()	Arbitrary redirect if URL is user-controllable
java.net.Socket	Network connection to arbitrary hosts if target is user-controlled

▸ ASP.NET — User Input Sources

Property	Description
Params / Item	Combined collection of query string, POST, cookies, and server variables
Form	POST form variables submitted by user
QueryString	URL query string parameters

Headers	HTTP request headers — attacker-controllable
Cookies	Cookie objects from the request
Url / RawUrl	Full request URL including query string
UrlReferrer	HTTP Referer header — attacker-controllable

▸ ASP.NET — Dangerous APIs

API / Class	Risk & Description
SqlCommand / OleDbCommand	SQL injection if CommandText contains concatenated user input
Process.Start()	OS command execution — path traversal attack if user controls part of filename
System.IO.File methods	37 methods all take filename — path traversal if user input is unsanitized
HttpResponse.Redirect()	Arbitrary redirect if URL string is user-controllable
Eval / Execute / ExecuteGlobal	Dynamic code execution — script injection if user data is incorporated
System.Net.Sockets.Socket	Network connection to arbitrary hosts if target is user-controlled

▸ PHP — User Input Sources

Variable	Description
$_GET / $_POST	Query string and POST body parameters respectively
$_COOKIE	Cookie values submitted in the request
$_REQUEST	Combined array of GET, POST, and COOKIE
$_FILES	Files uploaded in the request
$_SERVER['PHP_SELF']	Current script name — path info can be appended for XSS attacks
$_SERVER['HTTP_*']	All HTTP headers including custom ones — attacker-controllable
register_globals (if enabled)	Creates global variables for ALL request parameters — any variable name becomes user input!

▸ PHP — Dangerous APIs

API / Function	Risk & Description
include / require / include_once	File inclusion — if user controls filename, enables LFI/RFI and command execution
eval / create_function	Dynamic code execution — script injection if user data is passed in

exec / system / shell_exec / popen	OS command execution — commands chainable with | character
mysql_query / pg_query	SQL injection if user input is concatenated into query string
mysqli->prepare / bind_param	SAFE alternative — use parameter placeholders with bind_param()
fopen / file_get_contents	File access — supports HTTP/FTP URLs by default, enabling remote file inclusion
preg_replace with /e flag	Executes PHP code on every regex match — dangerous if user data is in the replacement

▸ Perl — Dangerous APIs

API / Function	Risk & Description
open()	If filename starts/ends with pipe character, contents are passed to command shell — command injection
eval	Dynamic execution of Perl code — script injection if user input is included
system / exec / qx / backticks	OS command execution — commands chainable with | character
selectall_arrayref / do	SQL injection if user input is in the query string

09 SECURITY-RELEVANT PLATFORM CONFIGURATION

Application security is significantly affected by platform configuration. Review these settings as part of every code review — misconfigurations are a major source of vulnerabilities.

▸ Java — web.xml Settings

Setting	Security Relevance
login-config	Authentication details — forms-based or Basic Auth. Verify j_security_check action and credential parameters
security-constraint	Defines which resources require authentication. Missing constraints expose unprotected endpoints
session-config / session-timeout	Session timeout in minutes. Short timeouts reduce hijacking window
error-page	Controls error responses. Verbose errors leak internal details to attackers
init-param: listings	Should be set to false — directory listings expose file structure
init-param: debug	Should be set to 0 — debug mode leaks sensitive information

▸ ASP.NET — Web.config Settings

Setting	Security Relevance
httpCookies: httpOnlyCookies	If true, cookies are HttpOnly — not accessible from client-side scripts
httpCookies: requireSSL	If true, cookies transmitted only over HTTPS — prevents interception
sessionState: timeout	Idle session expiration time in minutes
compilation: debug	If true, includes debug symbols — more verbose error info leaked to users
customErrors: mode	On or RemoteOnly shows generic errors. Off leaks detailed stack traces
httpRuntime: enableHeaderChecking	If true, checks headers for injection attacks including XSS
httpRuntime: enableVersionHeader	If true, outputs detailed ASP.NET version — helps attacker research
Connection strings	Should be encrypted using ASP.NET protected configuration feature

▸ PHP — php.ini Settings

Setting	Security Relevance
register_globals	If enabled, creates globals for ALL request params — severe security risk. Disabled by default since PHP 4.2
allow_url_fopen	If enabled (default), file functions can access remote URLs — enables RFI
allow_url_include	If enabled, file include functions can include remote files. Disabled by default in PHP 5.2+
display_errors	If enabled, PHP errors shown to users — information disclosure
magic_quotes_gpc	Auto-escapes quotes in input — does NOT prevent all SQL injection (numeric fields, second-order attacks)
safe_mode	Restricts some dangerous functions when enabled — but not comprehensive. Removed in PHP 6
file_uploads	If enabled, allows HTTP file uploads
upload_tmp_dir	Specifies temp directory for uploads — should not be world-readable

▸ Perl — Taint Mode

Perl provides a taint mode (#!/usr/bin/perl -T) that tracks user input and prevents tainted variables from being passed to dangerous functions like eval, system, exec, and open. To use tainted data, it must be 'cleaned' via pattern-matching and extraction of matched substrings. However, taint mode is only effective if developers use appropriate regex — if the expression is too liberal, the protection fails and the application remains vulnerable.

10 REGEX FUNDAMENTALS — From Beginner to Advanced

Regular Expressions (Regex) is a universal pattern matching language that works across all programming languages. In source code review, Regex is your most powerful tool for finding vulnerabilities automatically across thousands of files.

🔍 How Regex Helps in Code Review

Instead of reading every single line manually, Regex lets you search an entire codebase in seconds — finding hardcoded passwords, SQL injection points, insecure function calls, and exposed secrets instantly.

▸ Regex Syntax Reference

Symbol	Meaning	Example
`.`	`Any single character`	a.c → abc, adc, axc
`*`	`Zero or more of previous`	ab*c → ac, abc, abbc
`+`	`One or more of previous`	ab+c → abc, abbc
`?`	`Zero or one of previous`	ab?c → ac, abc
`^`	`Start of string`	^hello → "hello world"
`$`	`End of string`	world$ → "hello world"
`[]`	`Character class`	[abc] → a, b, or c
`[^]`	`Negated character class`	[^abc] → not a, b, or c
`\d`	`Any digit (0-9)`	\d+ → 123, 4567
`\w`	`Word char (a-z,A-Z,0-9,_)`	\w+ → hello_123
`\s`	`Whitespace character`	\s+ → space, tab
`{n}`	`Exactly n repetitions`	\d{3} → 123
`{n,m}`	`Between n and m times`	\d{2,4} → 12, 123, 1234
`()`	`Grouping`	(ab)+ → ab, abab
`|`	`OR (alternation)`	cat|dog → cat or dog
`(?:)`	`Non-capturing group`	(?:ab)+ → groups without capture
`(?=)`	`Positive lookahead`	\d(?=px) → digit before px
`(?!)`	`Negative lookahead`	\d(?!px) → digit NOT before px

11 REGEX IN EVERY PROGRAMMING LANGUAGE

Regex syntax is nearly identical across languages, but the implementation method differs. Here is how to use Regex in each language for security-focused code review.

▸ Python

```python
import re

# Search for hardcoded passwords
pattern = r'(password|passwd|pwd)\s*[:=]\s*["\'][^"\']+["\']'
matches = re.findall(pattern, code_text)

# Search for dangerous eval/exec calls
dangerous = re.findall(r'\b(eval|exec|os\.system)\s*\(', code_text)

# Validate email format
email_pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
is_valid = re.match(email_pattern, user_email)
```

▸ JavaScript / TypeScript

```javascript
// Search for hardcoded API keys
const pattern = /api[_-]?key\s*[=:]\s*["'`][A-Za-z0-9-_]+["'`]/gi;
const matches = sourceCode.match(pattern);

// Find SQL concatenation patterns
const sqli = sourceCode.match(/["']\s*(SELECT|INSERT|DELETE).*\+/gi);

// Replace sensitive tokens for safe logging
const clean = code.replace(/Bearer\s+[A-Za-z0-9._-]+/g, 'Bearer REDACTED');
```

▸ Java

```java
import java.util.regex.*;

// Search for dangerous Runtime.exec() calls
Pattern p = Pattern.compile("Runtime.*exec\\s*\\(");
Matcher m = p.matcher(sourceCode);
while (m.find()) {
    System.out.println("Command exec at: " + m.group());
}
```

```
// Find createStatement usage (potential SQLi)
Pattern sqli = Pattern.compile("createStatement|executeQuery.*\\+");
```

▸ PHP

```
// Find file inclusion with user variables
$pattern = '/(require|include)(_once)?\s*\(\s*\$/i';
preg_match_all($pattern, $phpCode, $includes);

// Find eval/exec usage
$dangerous = '/(eval|exec|system|shell_exec|popen)\s*\(/i';
preg_match_all($dangerous, $phpCode, $exec_matches);

// Find register_globals variable usage
$globals = '/\$[a-zA-Z_]+\s*=.*\$_?(GET|POST|REQUEST)/i';
```

▸ Go, Ruby, C#, Swift, Kotlin

```
// Go — Find SQL concatenation
re := regexp.MustCompile(`"\s*(SELECT|INSERT).*" \+`)
matches := re.FindAllString(sourceCode, -1)

// Ruby — Find dangerous eval/exec
pattern = /\b(eval|exec|system|`[^`]*`)\s*/i
matches = source_code.scan(pattern)

// C# — Find Process.Start with string concat
var re = new Regex(@"Process\.Start\s*\(", RegexOptions.IgnoreCase);
var matches = re.Matches(sourceCode);
```

12 REGEX PATTERNS FOR SECURITY CODE REVIEW

These are ready-to-use Regex patterns specifically designed to find common vulnerabilities in source code. Use these in your code review workflow.

▸ Finding Hardcoded Secrets

Target	Regex Pattern	Finds
Passwords	`(password|passwd|pwd)\s*[:=]\s*["'][^"']+["']`	password = "abc123"
API Keys	`(api[_-]?key|apikey)\s*[:=]\s*["'][A-Za-z0-9-_]+["']`	api_key = "sk-live-xxx"
AWS Keys	`AKIA[0-9A-Z]{16}`	AKIAIOSFODNN7EXAMPLE
JWT Tokens	`eyJ[A-Za-z0-9_-]*\.eyJ[A-Za-z0-9_-]*\.[A-Za-z0-9_-]*`	eyJhbGciOi...
Private Keys	`-----BEGIN (RSA |EC )?PRIVATE KEY-----`	-----BEGIN PRIVATE KEY-----
Tokens	`(token|secret)\s*[:=]\s*["'][A-Za-z0-9._-]{20,}["']`	token = "abc...xyz"

▸ Finding Injection Vulnerabilities

Vulnerability	Regex Pattern	Detects
SQL Injection	`".*\+.*\$(\w+).*WHERE`	"SELECT * " + id + " WHERE"
Command Injection	`(system|exec|popen|shell_exec)\s*\(`	system(userInput)
Path Traversal	`(\.\./|\.\.\\)`	../../etc/passwd
eval() Usage	`(eval|exec|Function)\s*\(`	eval(userInput)
File Inclusion	`(require|include)\s*\(\s*\$`	include($file)
Insecure Deserialize	`(unserialize|pickle\.loads|ObjectInputStream)`	pickle.loads(data)

▸ Finding Insecure Configurations

Issue	Regex Pattern	Catches
HTTP not HTTPS	`http://[^\s"'<>]+(api|login|pay)`	http://api.example.com
Weak Hash	`(md5|sha1)\s*\(`	md5(password)

Debug Mode	`(DEBUG|debug)\s*[:=]\s*(true|True|1)`	DEBUG = True
Commented Secrets	`//.*?(password|secret|key)\s*[:=]`	// password = "old"
Security TODOs	`(TODO|FIXME|HACK).*?(security|auth|password)`	// TODO: fix auth
SSL Disabled	`verify\s*=\s*False|rejectUnauthorized.*false`	verify=False

13 ADVANCED PYTHON SCANNER — Automated Code Review Tool

This advanced Python script automates source code review by scanning entire project directories for vulnerabilities using Regex patterns. It supports multiple languages, generates severity-rated reports with CWE mapping, and outputs findings in structured format.

```
#!/usr/bin/env python3
"""
Advanced Source Code Security Scanner
Author: 0x2nac0nda
Description: Automated source code review tool using Regex
             to detect vulnerabilities across all languages.
"""
import re, os, sys, json, argparse
from datetime import datetime
from collections import defaultdict
from pathlib import Path

SUPPORTED_EXTENSIONS = {
    '.py', '.js', '.ts', '.java', '.php', '.cs', '.go',
    '.rb', '.swift', '.kt', '.c', '.cpp', '.h', '.jsx', '.tsx', '.pl'
}
EXCLUDED_DIRS = {
    'node_modules', '.git', 'venv', '__pycache__',
    '.env', 'dist', 'build', 'vendor'
}
PATTERNS = {
  'CRITICAL': {
    'Hardcoded Password':     { 'regex': r'(password|passwd|pwd)\s*[:=]\
s*["\'][^"\']{3,}["\']',
                                'desc': 'Password hardcoded in source', 'cwe':
'CWE-259' },
    'Hardcoded API Key':      { 'regex': r'(api[_-]?key|apikey|api_secret)\
s*[:=]\s*["\'][A-Za-z0-9-_]{10,}["\']',
                                'desc': 'API key exposed in source', 'cwe':
'CWE-798' },
    'AWS Access Key':         { 'regex': r'AKIA[0-9A-Z]{16}',
                                'desc': 'AWS Access Key ID found', 'cwe':
'CWE-798' },
    'Private Key Exposed':    { 'regex': r'-----BEGIN\s*(RSA\s+|EC\s+)?
PRIVATE\s+KEY-----',
                                'desc': 'Private key in source code', 'cwe':
'CWE-798' },
    'JWT Secret':             { 'regex': r'(jwt|token).*(secret|key)\s*[:=]\
s*["\'][^"\']{8,}["\']',
                                'desc': 'JWT secret key hardcoded', 'cwe':
'CWE-798' }
  },
  'HIGH': {
    'SQL Injection':          { 'regex': r'(SELECT|INSERT|UPDATE|
DELETE).*["\']\s*\+\s*\w+',
                                'desc': 'SQL injection via string concat',
'cwe': 'CWE-89' },
    'Command Injection':      { 'regex': r'(os\.system|subprocess\.call|exec|
shell_exec|popen)\s*\(',
                                'desc': 'Dangerous command execution', 'cwe':
'CWE-78' },
    'Eval/Exec Usage':        { 'regex': r'\b(eval|exec|Function)\s*\(',
                                'desc': 'Dynamic code execution', 'cwe': 'CWE-
94' },
    'Insecure Deserialize':   { 'regex': r'(pickle\.loads|unserialize|
ObjectInputStream|Marshal\.load)',
                                'desc': 'Unsafe deserialization', 'cwe': 'CWE-
502' },
    'Path Traversal':         { 'regex': r'(\.\./|\\\.\\.)',
                                'desc': 'Path traversal detected', 'cwe':
'CWE-22' },
    'File Inclusion':         { 'regex': r'(require|include)(_once)?\s*\(\s*\
$',
                                'desc': 'Dynamic file inclusion (LFI/RFI)',
'cwe': 'CWE-98' }
  },
  'MEDIUM': {
    'Weak Hash':              { 'regex': r'\b(md5|sha1)\s*\(',
                                'desc': 'Weak cryptographic hash', 'cwe':
'CWE-327' },
    'HTTP Not HTTPS':         { 'regex': r'http://[^\s"\']+((api|login|auth|
pay|admin))',
                                'desc': 'Insecure HTTP for sensitive
endpoint', 'cwe': 'CWE-319' },
    'SSL Verify Off':         { 'regex': r'(verify\s*=\s*False|
rejectUnauthorized.*false)',
                                'desc': 'SSL certificate verification
disabled', 'cwe': 'CWE-295' },
    'Sensitive Data Logged':  { 'regex': r'(print|log|console).*(password|
secret|token|key)',
                                'desc': 'Possible sensitive data in logs',
'cwe': 'CWE-532' }
  },
  'LOW': {
    'Debug Mode':             { 'regex': r'(DEBUG|debug)\s*[:=]\s*(true|True|
TRUE|1)',
                                'desc': 'Debug mode enabled', 'cwe': 'CWE-
94' },
    'Commented Creds':        { 'regex': r'(//|#|/\*).*?(password|secret|
api_key)\s*[:=]',
                                'desc': 'Commented-out credentials', 'cwe':
'CWE-798' },
    'Security TODO':          { 'regex': r'(TODO|FIXME|HACK|XXX).*(security|
auth|password|vuln)',
                                'desc': 'Security-related TODO found', 'cwe':
'CWE-120' }
  }
}
class SecurityScanner:
    def __init__(self, target_path):
        self.target_path   = Path(target_path)
        self.findings      = []
        self.stats         = defaultdict(int)
        self.scanned_files = 0

    def is_valid_file(self, fp):
        if fp.suffix not in SUPPORTED_EXTENSIONS: return False
        return not any(ex in fp.parts for ex in EXCLUDED_DIRS)

    def scan_file(self, filepath):
        try:
            lines = open(filepath, 'r', errors='ignore').readlines()
        except: return
        self.scanned_files += 1
        for severity, pats in PATTERNS.items():
            for name, cfg in pats.items():
                compiled = re.compile(cfg['regex'], re.IGNORECASE)
                for ln, line in enumerate(lines, 1):
                    if compiled.findall(line):
                        self.findings.append({
                            'severity': severity, 'vulnerability': name,
                            'file': str(filepath), 'line': ln,
                            'code': line.strip(), 'desc': cfg['desc'],
                            'cwe': cfg['cwe'] })
                        self.stats[severity] += 1
                        break
    def scan_project(self):
        print(f"[*] Scanning: {self.target_path}")
        print(f"[*] Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
        for fp in self.target_path.rglob('*'):
            if fp.is_file() and self.is_valid_file(fp):
                self.scan_file(fp)
        self.findings.sort(
            key=lambda x:
['CRITICAL','HIGH','MEDIUM','LOW'].index(x['severity']))

    def print_report(self):
        colors = {'CRITICAL':'\033[91m','HIGH':'\033[93m',
                  'MEDIUM':'\033[94m','LOW':'\033[92m'}
        R = '\033[0m'
        print("\n" + "=" * 60)
        print("  SOURCE CODE SECURITY SCAN REPORT")
        print(f"  Author: 0x2nac0nda | Files: {self.scanned_files}")
        print(f"  Total: {len(self.findings)} | Crit:
{self.stats['CRITICAL']}")
        print(f"  High: {self.stats['HIGH']} | Med: {self.stats['MEDIUM']}")
        print("=" * 60)
        for f in self.findings:
            s = f['severity']
            print(f"\n{colors[s]}[{s}]{R} {f['vulnerability']}")
            print(f"  File: {f['file']} | Line: {f['line']} | {f['cwe']}")
            print(f"  Info: {f['desc']}")
            print(f"  Code: {f['code'][:100]}")
    def export_json(self, out):
        report = {
            'scanner': 'Source Code Security Scanner',
            'author': '0x2nac0nda',
            'timestamp': datetime.now().isoformat(),
            'target': str(self.target_path),
            'summary': { 'files': self.scanned_files,
                'total': len(self.findings),
                'critical': self.stats['CRITICAL'],
                'high': self.stats['HIGH'],
                'medium': self.stats['MEDIUM'],
                'low': self.stats['LOW'] },
            'findings': self.findings }
        json.dump(report, open(out,'w'), indent=4)
        print(f"\n[+] JSON report saved: {out}")

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Security Scanner')
    parser.add_argument('path', help='Target directory')
    parser.add_argument('--json', help='Export JSON report', default=None)
    args = parser.parse_args()
    s = SecurityScanner(args.path)
    s.scan_project()
    s.print_report()
    if args.json: s.export_json(args.json)

# Usage: python scanner.py ./project
# Usage: python scanner.py ./project --json report.json
```

14 COMMON VULNERABILITY CHECKLIST

Use this checklist during every source code review. Check each item systematically.

Checklist Item	What to Verify
SQL Injection	No string concatenation in DB queries — use parameterized queries or ORM
XSS	All user output is properly encoded before rendering in HTML or JS
CSRF Protection	All state-changing endpoints have CSRF token validation
SSRF Prevention	URL inputs validated against whitelist — no internal service access
Path Traversal	File paths are sanitized and canonicalized before use
Command Injection	No user input in system/exec calls — use safe argument arrays
Insecure Deserialize	No deserialization of untrusted data (pickle, Marshal, ObjectInputStream)
XXE Prevention	XML parsers have DTD processing and external entities disabled
IDOR / BOLA	Server-side ownership check before returning or modifying any object
Access Control	Every endpoint has proper authorization middleware and role checks
Mass Assignment	Only whitelisted fields accepted from user input
Info Disclosure	Error messages are generic — no stack traces in production
File Upload	File type validated by magic bytes — uploads stored outside web root
Weak Crypto	Only strong algorithms (AES-256, bcrypt, argon2) — no MD5 or SHA1
Hardcoded Secrets	No credentials or API keys in source code — use env variables
Security Headers	HSTS, CSP, X-Frame-Options, X-Content-Type-Options configured
Cookie Security	All cookies have Secure, HttpOnly, and SameSite flags
Rate Limiting	Auth and sensitive endpoints have rate limiting
Audit Logging	All security-critical operations are logged with user context
Arbitrary Redirect	All redirect URLs validated against whitelist of allowed domains
Backdoor Check	Credential validation logic has no hardcoded bypass passwords

Comment Audit	Search codebase for: bug, todo, fix, hope, hack, overflow, inject, xss

15 REPORTING & DOCUMENTATION

Professional reporting is the final and critical step. A well-documented report demonstrates expertise and provides actionable remediation for development teams.

▸ Report Structure

- Executive Summary — high-level overview of findings and risk level
- Scope & Methodology — what was reviewed, which approach (black/white-box), tools used
- Findings Summary — count by severity with visual breakdown
- Detailed Findings — each vulnerability with full documentation
- Remediation Guide — secure code fixes for each finding
- Compliance Mapping — map findings to OWASP, CWE, PCI-DSS
- Appendix — tools used, references, and raw data

▸ Each Finding Must Include

Severity	Critical / High / Medium / Low / Informational
Vulnerability Name	Clear, descriptive name of the vulnerability type
File & Line Number	Exact location in the codebase
Vulnerable Code	The exact code snippet that contains the vulnerability
Description	Technical explanation of why this code is vulnerable
Impact	What an attacker could achieve by exploiting this
CVSS Score	Common Vulnerability Scoring System rating
CWE Reference	CWE ID linking to the weakness enumeration database
OWASP Category	Which OWASP Top 10 category this finding belongs to
Proof of Concept	Example showing how the vulnerability can be exploited
Remediation	Secure code fix — the corrected version of vulnerable code
Priority	Order in which findings should be addressed based on risk

▸ Severity Classification Guide

Critical	Complete system compromise. RCE, hardcoded credentials, backdoor passwords, authentication bypass with full access
High	Significant security impact. SQL injection, command injection, access to other users' data, session hijacking, arbitrary redirection

Medium	Moderate impact. XSS without session access, info disclosure, missing rate limiting, weak cryptography, path traversal with limited scope
Low	Minor impact. Debug mode, commented credentials, security TODOs, missing headers, verbose errors in non-production
Info	Best practice not followed. Code quality issues, potential future risks, recommendation improvements

▸ Output Formats

- PDF Report — professional document for stakeholders and management
- JSON Export — structured data for CI/CD pipeline integration
- CSV Export — spreadsheet format for tracking and prioritization
- HTML Report — interactive report with filtering and sorting
- Markdown — documentation format for GitHub or internal wikis

16 QUICK REFERENCE — Regex Cheat Sheet

▸ Character Classes

Pattern	Matches	Example
`\d`	Digit 0-9	\d+ → 123
`\D`	Not a digit	\D+ → abc
`\w`	Word character	\w+ → hello_1
`\W`	Not word character	\W+ → !@#
`\s`	Whitespace	\s+ → space/tab
`\S`	Not whitespace	\S+ → hello
`[a-z]`	Lowercase letter	[a-z]+ → abc
`[A-Z]`	Uppercase letter	[A-Z]+ → ABC
`[0-9]`	Digit (same as \d)	[0-9]+ → 456
`.`	Any character	.+ → anything

▸ Quantifiers

Pattern	Meaning	Example
`*`	Zero or more	ab* → a, ab, abb
`+`	One or more	ab+ → ab, abb
`?`	Zero or one	ab? → a, ab
`{n}`	Exactly n times	a{3} → aaa
`{n,}`	n or more times	a{2,} → aa, aaa
`{n,m}`	Between n and m	a{2,4} → aa to aaaa
`*?`	Lazy zero or more	a*? → minimum match
`+?`	Lazy one or more	a+? → minimum match

▸ Anchors & Boundaries

Pattern	Meaning	Example
`^`	Start of string	^Hello → "Hello World"
`$`	End of string	World$ → "Hello World"
`\b`	Word boundary	\bcat\b → "the cat sat"
`\B`	Not word boundary	\Bcat → "concatenate"

▸ Groups & Alternation

Pattern	Meaning	Example
`(abc)`	`Capturing group`	(ab)+ → ab, abab
`(?:abc)`	`Non-capturing group`	(?:ab)+ → no capture
`|`	`Alternation (OR)`	cat|dog → cat or dog
`(?=abc)`	`Positive lookahead`	\d(?=px) → digit before px
`(?!abc)`	`Negative lookahead`	\d(?!px) → not before px
`(?<=abc)`	`Positive lookbehind`	(?<=\$)\d+ → after $
`(?<!abc)`	`Negative lookbehind`	(?<!\$)\d+ → not after $

17 TIPS FOR VISUAL STUDIO CODE EDITOR

VS Code is one of the best editors for source code review. These tips will make your security-focused code review faster, more accurate, and more efficient.

▸ Essential Extensions to Install

Semgrep	Scans your code in real-time for security vulnerabilities using community rules — highlights issues as you type
Snyk Security	Detects vulnerabilities in your code AND dependencies simultaneously — shows severity inline
GitLens	Shows who wrote every line and when — critical for understanding code history and finding suspicious changes
ESLint / Pylint	Enforces code quality and flags insecure patterns for JavaScript and Python respectively
Remote SSH	Review code on remote servers directly inside VS Code — no need to copy files locally
Docker	Browse and edit files inside containers — useful for reviewing containerized applications
YAML / JSON / XML	Syntax highlighting and validation for config files where secrets often hide
Thunder Client	Built-in HTTP client — test API endpoints during your review without leaving the editor

▸ Search & Replace Shortcuts

VS Code search is your fastest weapon during code review. Master these shortcuts to scan thousands of lines in seconds.

Ctrl + H (Find & Replace)	Open the search panel — type your regex pattern in the search box
Alt + R (Regex Toggle)	Click the .* button or press Alt+R to enable regex mode in search
Ctrl + Shift + F (Search All Files)	Search across the ENTIRE project — not just the current file
Alt + Enter (Select All Matches)	Highlights every match in the project — see all instances at once
Ctrl + G (Go To Line)	Jump directly to any line number — useful after finding a match in results
Ctrl + D (Select Next Match)	Selects the next occurrence of the current word — great for tracing a variable

F12 (Go To Definition)	Jump to where a function or variable is defined — trace data flow instantly
Shift + F12 (Find All References)	Find every place a function or variable is used — critical for tracing user input flow

▸ Ready-to-Use Search Patterns for VS Code

Copy these directly into VS Code search box with Regex mode (Alt+R) enabled. Each targets a specific vulnerability class.

```
# ─── Enable Regex mode (Alt+R) then paste these ───

# Find hardcoded passwords
(password|passwd|pwd)\s*[:=]\s*["'][^"']{3,}["']

# Find eval / exec calls
\b(eval|exec|Function)\s*\(

# Find SQL string concatenation
(SELECT|INSERT|UPDATE|DELETE).*["']\s*\+\s*\w+

# Find commented-out credentials
(//|#|/\*).*?(password|secret|api_key)\s*[:=]

# Find AWS access keys
AKIA[0-9A-Z]{16}

# Find security TODOs
(TODO|FIXME|HACK).*(security|auth|password)

# Find debug mode enabled
(DEBUG|debug)\s*[:=]\s*(true|True|1)
```

▸ Workspace & File Management Tips

- Use Workspaces — open multiple project folders at once to compare code across services
- Exclude node_modules and .git in settings.json so project-wide search stays fast
- Enable the Minimap (View > Minimap) — quickly scroll through large files visually
- Pin important files — right-click a tab and pin it so it stays open while you navigate
- Split Editor (Ctrl + \\) — compare vulnerable code side-by-side with the secure fix
- Sticky Scroll (View > Sticky Scroll) — keeps the current function header visible while scrolling down
- Problems Panel (Ctrl + Shift + M) — see all linter warnings and errors in one single place

▸ Debugging & Tracing User Input

VS Code built-in debugger is powerful for tracing how user input flows through an application — the core technique of white-box testing.

- Set breakpoints on every function that receives user input — step through and watch data transform
- Use the Watch panel — add variables to monitor their values as execution progresses
- Run and Debug (Ctrl + Shift + D) — attach to a running application to debug live
- Integrated Terminal (Ctrl + `) — run scanner.py or grep commands without leaving VS Code
- Use launch.json configs — save debug profiles for each project so you can restart instantly

18 KEY TAKEAWAYS & NEXT STEPS

✅ Source Code Review

Combine black-box and white-box approaches. Follow a structured methodology: recon → scan → review → classify → report. Trace user input through the code, search for vulnerability signatures, and review risky areas line-by-line. Always verify prima facie vulnerabilities on the running application.

✅ Regex Mastery

Your most powerful tool for automating code review. Master character classes, quantifiers, anchors, and groups. Use it to scan for hardcoded secrets, injection patterns, and insecure configurations. Every language supports the same core syntax.

✅ Platform Awareness

Every platform has unique dangerous APIs and configuration settings. Java's Runtime.exec(), PHP's include/register_globals, ASP.NET's Process.Start(), Perl's open() with pipes — know the platform-specific risks for the language you are reviewing.

▸ Next Steps

- Practice on vulnerable apps: DVWA, Juice Shop, WebGoat
- Learn Semgrep — write custom rules for your target language
- Practice Regex daily on regex101.com
- Review real CVEs and trace them back to vulnerable code patterns
- Use the Advanced Python Scanner on real projects
- Study OWASP ASVS for comprehensive security verification
- Practice writing secure code fixes alongside finding vulnerabilities
- Build a personal security pattern cheat sheet for quick reference

🛡️ **Source Code Review & Regex Guide**

Written by **Abdalla Abdelrhman**

Cybersecurity Consultant

🔗 linkedin.com/in/0x2nac0nda