Beagle Security

Software security audit checklist

A practical companion piece to our software security audit guide. Use it to scope the audit, review the application and its environment, test the running system, and keep remediation moving.

Work through each section with your engineering, security, and product stakeholders. Tick items when the evidence is available, not when the policy merely says it should be.

Before the audit

- [ ] **Define the audit scope and objectives**

  List the applications, APIs, environments, integrations, and business processes in scope. Be specific about what the audit needs to answer, whether that is release readiness, a compliance requirement, or a broader security review.

- [ ] **Identify crucial assets and data flows**

  Map the customer data, credentials, payment information, and other sensitive assets the application handles. Trace where that data enters, moves, and is stored so the audit does not focus only on the front end.

- [ ] **Choose the audit types that fit the application**

  Use the architecture and risk profile to decide what is needed: code review, configuration and infrastructure review, penetration testing, dependency review, or a compliance audit. One type alone rarely gives the full picture.

- [ ] **Gather relevant documentation and environment details**

  Collect architecture diagrams, data-flow diagrams, asset inventories, deployment details, access-control documentation, recent test results, and known exceptions. Outdated documentation can create blind spots before testing even starts.

- [ ] **Set owners, timelines and decision points**

  Confirm who provides access, who reviews findings, who approves risk acceptance, and who owns remediation. An audit without named owners quickly becomes a report that nobody acts on.

Review code, dependencies and design

- [ ] **Review source code for insecure patterns**

  Use SAST and manual review to look for weak input validation, insecure authentication or authorization logic, unsafe cryptography, and other coding issues. Focus additional manual attention on the parts of the application that handle sensitive actions.

- [ ] **Check business logic and authorization paths**

  Test whether users can access data, perform actions, or move through workflows they should not be able to. These issues often sit in the way features work together, not in a single vulnerable line of code.

- [ ] **Audit third-party libraries and packages**

  Inventory direct and transitive dependencies, identify known CVEs, and flag unsupported or outdated packages. A dependency that has not changed in your codebase can still become risky when new vulnerabilities are disclosed.

- [ ] **Review secrets and sensitive configuration**

  Check that API keys, credentials, certificates, and tokens are not committed to code or exposed through build logs and configuration files. Confirm that secrets are stored and rotated through an approved process.

- [ ] **Assess secure development practices**

  Confirm developers use secure coding guidance, code review includes security considerations, and security testing happens early enough to influence design decisions instead of only releasing sign-off.

Review configurations and infrastructure

- [ ] **Assess cloud, server, and container configurations**

  Check for overly permissive IAM roles, unnecessary services, exposed management interfaces, weak TLS settings, and storage that is publicly accessible when it should not be.

- [ ] **Review network exposure and segmentation**

  Confirm only required ports and services are reachable, production is separated from lower environments, and internal services are not unintentionally exposed to the public internet.

- [ ] **Verify identity and access controls**

  Review user lifecycle processes, least-privilege access, privileged-account controls, multi-factor authentication, and session management. Pay particular attention to cloud consoles, CI/CD systems, and admin interfaces.

- [ ] **Assess database and data-store security**

  Confirm databases use restricted service accounts, sensitive fields are encrypted where appropriate, backups are protected, and access logs are available for important administrative activity.

- [ ] **Check change and patch management**

  Verify critical patches have defined remediation windows, production changes go through review, and teams can roll back unsafe changes. Look at the oldest open critical issue, not just the policy.

Test the running application

- [ ] **Run authenticated penetration testing**

  Test the application as the relevant user roles so authenticated pages, workflows, and authorization controls are assessed. Public scanning alone cannot show how the application behaves after login.

- [ ] **Validate authentication and session behavior**

  Test login, logout, password-reset, multi-factor authentication, token expiry, and session revocation. Confirm the controls still work under normal and edge-case user journeys.

- [ ] **Test application and API attack surface**

  Assess the live application for common vulnerabilities, exposed endpoints, insecure API behavior, and configuration weaknesses. Include the APIs and integrations that support the main product workflows.

- [ ] **Record complex workflows where needed**

  Identify business logic flows that crawling may not fully cover, such as approvals, transfers, claims, account changes, or role changes. Record those scenarios so testing can reach the relevant parts of the application.

- [ ] **Review evidence and reproduce material findings**

  For high-impact issues, confirm the affected asset, required conditions, and business impact. Clear, reproducible evidence helps engineering teams fix the right problem without spending time on ambiguity.

Report, remediate and maintain

- [ ] **Prioritize findings by risk and business impact**

  Use severity scoring alongside the sensitivity of the affected asset, exploitability, and likely business impact. A technically moderate issue can still need urgent attention if it exposes customer or regulated data.

- [ ] **Create a remediation plan with owners and dates**

  Every accepted finding should have a named owner, a target date, and a clear next step. Separate immediate containment from long-term fixes so urgent risks do not wait for a larger engineering project.

- [ ] **Document risk acceptance decisions**

  When a risk cannot be fixed immediately, record why, who approved it, what compensating controls exist, and when the decision will be reviewed. Risk acceptance should be a decision, not an abandoned ticket.

- [ ] **Communicate findings to technical and business teams**

  Prepare an executive summary for leadership and actionable technical detail for the people doing the work. The same report should not force every audience to extract the information they need.

- [ ] **Retest completed fixes**

  Verify that remediated findings are actually resolved and that the change did not introduce a related issue. A closed ticket is not the same thing as a verified fix.

- [ ] **Build the audit into ongoing security work**

  Set review cycles based on application changes and risk, monitor the environment continuously, and reassess scope when features, integrations, or data flows change. The audit should improve the next release, not only document the previous one.

Put the application review into practice

Beagle Security helps teams run continuous, authenticated penetration tests against web applications and APIs. Use the results alongside your audit evidence to validate fixes and keep security checks aligned with how the application changes.