SafeStack

Please send feedback to hello@safestack.io

Revised: **2020-02-20**

Code Review Security Checklist Implementation Manual

Introduction

The Checklist was created to improve security culture in dev teams and help them consistently check their code for common security risks. The earlier that vulnerabilities are discovered, the cheaper and easier they are to fix. The tool is intended for use as part of a software team's code review process. Our hope is that it will improve teams' overall security posture and the quality of the code they release.

How to use this manual

The "reviewing team" is understood to comprise the original author, a primary reviewer who approves the change, testers, business analysts, and any secondary reviewers that are tagged in. Everyone in the reviewing team plays a role in enabling safe code reviews.
This manual provides guidance on using the checklist, suggestions for implementation, and explanations of each entry. Teams should adapt the checklist to their own circumstances. Each check has been included based on expert opinion that its inclusion will reduce the likelihood of security risks and that it is unlikely to introduce risk to a system or add unmanageable cost.
The ultimate goal of the Checklist is to prompt a security mindset at code review time and to make it safer and easier to discuss possible security issues in code.

How to run the Checklist (in brief)

The Checklist has three phases - before code is pushed, during the code review, and before the code review is marked complete. The Checklist itself can be included as a template in a code review request and the review tools configured to require its completion.

The first phase takes place *before* the original author shares their code with the team and consists of the author verifying they haven't included any real passwords, keys, tokens, or other secrets in their code.

The next phase happens during review and each item may be completed by any of the reviewers besides the original author. The reviewers confirm the right people have been tagged in and that they all understand the intended change.

They then check for the presence of debug code, the handling of untrusted data and response information, the correct use of tools, and that there is sufficient log and test coverage.

Finally, if the code review has raised risks beyond the scope of the review to fix, the reviewers raise the risk to their team and ensure it is logged somewhere it will be reviewed. This can also be completed by any of the reviewers

How to run the Checklist (in detail)

Before pushing code to the team repository

These checks are to be completed by the author of the change before any code is pushed to a shared area. If continuously integrating that may mean performing this check with every push.

Have all secrets been removed from the committed code?

Once code has been shared, it is annoying and error-prone to remove passwords, keys, certificates and other secrets from a branch of commits. It can also be difficult to review, as a secret might have been added in an early commit and removed in a subsequent one. Secrets that do make it into a shared repository should be replaced. It may be possible to automatically catch some secrets (by e.g. searching for variables named "password", but only the author can really confirm they haven't included sensitive information.
Tests may contain secrets only if they are set and consumed by the tests themselves. Prefer generated secrets to hand-created ones. Clearly mark fake secrets as such to make it easier to review.

During a review of the code

These checks are completed during the review of the code by the primary reviewer in order to confirm safety before merging into a shared branch.

Have the right people been engaged to review the code?

Choosing an effective group of reviewers can be tough as the team needs to balance their workload with the desire to have everyone's eyes on changes. Aim for two-to-three reviewers for most changes, and tag in additional people when the change:

- Involves deeper knowledge of the system,
- Interacts with another team's system,
- Could benefit from other specialised knowledge e.g. SME's, security folks for changes to authentication or authorisation components,
- Is considered "risky" or "tricky".

Consider including people who are expected to test the change, and folks less familiar with the system who can ask outsider questions.

Is the purpose of the change stated and understood by the reviewers?

Everyone should have a clear grasp of what the change is trying to do. Teams with issue tracking systems should require all proposed changes to reference the related issue and vice versa. The one irreplaceable feature of manual code reviews is judgement; does this change do what is intended? Be on the lookout also for functionality that isn't related to the core purpose. While a refactor that enables the change is fine, additional changes and fixes should really have their own review. At the very least, they should be highlighted in the change's summary.

Is there debug functionality in the code?

Sometimes we write code that makes our system easier to debug; by emitting extra information or enabling actions outside of the normal workflow. It's easy to forget to remove this code before pushing it into production. Always remove debug code if possible. If debug code is needed it must be guarded to prevent it running outside of specific test environments, and this check should fail closed. Consider writing tests that confirm this.

Do log entries cover all key events and states?

Investigating performance and security issues is greatly assisted by systems that emit helpful logs. At a minimum, log: process start, process end, any exceptions thrown, audit events, input and output events.

Do log entries include enough information to uniquely identify the event?

Events such as user requests should generate a unique identifier that is included in all related messages to enable correlation of individual requests across multiple components. This is critical to understanding how an event plays across the entire system. Include high cardinality fields in log events, and consider using structured logging.

Do log entries exclude secrets and customers' PII?

As logs by their nature cross trust boundaries, they should limit the amount of customer information that they record - ideally to identifying tokens. Where PII logging is necessary for debugging, care should be taken to limit these entries to diagnostic tools rather than auditing or archival stores.

Do response messages make use of appropriate status codes? Do response messages exclude information that should remain internal to the system? Do response messages limit information to the correct level of authorization?

Clear, consistent status codes enable the team to monitor operational issues quickly and effectively. Specifically, error responses that differentiate between auth* failures, server errors and client errors can be vital. That said, it's important not to accidentally leak information by e.g.

returning a different status code if a record exists than if it doesn't if the user is not entitled to know this. All sensitive responses should pass through access controls that prevent unauthorized leaking of information.

Is user-supplied data validated before it is used or stored? Is user-supplied data escaped to suit the context in which it is interpreted?

Injection attacks remain an evergreen threat wherever data is passed to an interpreter e.g. a browser, database query engine, shell etc. Any untrusted data should be validated on receipt and be escaped before being used in an interpreter.

Are dependencies being used effectively?

Frameworks often provide tools to defend against injection attacks, to provide authentication and access control, to emit logs etc; the team must be familiar with how to use these tools safely, and identify when they are making a risky choice.

Have new dependencies been vetted? Are they up-to-date?

Supply-chain attacks are becoming increasingly common and out-of-date dependencies pose a major risk if they have known vulnerabilities. The team should have processes for assessing new dependencies and for keeping their ecosystem current. In general, software composition analysis tools can (and should) be integrated into build pipelines to automate identification of vulnerable dependencies. Use judgement and coordination to decide *when* to update a version.

To testers: is the test coverage sufficient? Are misuse cases represented?

Tests can identify failing edge cases, bug regressions and more. While the developer may have written low-level and even integration tests, a tester's perspective is crucial to identifying risks and unexpected inputs. A misuse case is a representation of something that should not happen and should be tested for. While test cases should be identified earlier in the development process, this is the last chance to check if they are missing, or if further manual testing is needed.

Before completing the code review

This check takes place immediately before completing the review as concerns that are mitigated or found not to qualify as risks need not be escalated beyond the review.

Confirm unresolved risks have been raised and documented.

It may not be possible to fully mitigate an identified risk during review. The risk may even be integral to the nature of the change. Given this, identified risks should be communicated to the wider team and formally accepted by the system owners.

Additional notes

Modifying the Checklist

Teams should modify the Checklist to account for differences in terminology, technological platforms, and team composition. They shouldn't remove safety steps because they are unable or unwilling to perform them.

The entire team should be involved in decisions to modify the Checklist, and the modified Checklist tested on a single system to ensure it works as intended.

The key principles to take into account when modifying the Checklist are:

- **Focused** on critical issues not adequately covered by other controls.
- **Quick** to run through.
- Have clearly **Actionable** items to prevent confusion.
- Have a **Collaborative** ownership.
- **Tested** in small increments.
- **Integrated** into the team's workflow.

Teams may consider adding extra safety checks, especially if they are needed in their domain. They should be cautious not to make the Checklist too complex or onerous.

Introducing the Checklist to a code review process

Leadership from senior developers needs to be demonstrated if the Checklist is to succeed.

- Identify and work with a core of people who are enthusiastic about the Checklist, trying to include members outside of development and business leaders.
- Start small, with one team and one system. Wait until they are comfortable with the process and have worked out the kinks.
- Track changes and improvements to metrics such as vulnerabilities that make it to release.

It is possible to include the checklist items in a code review template (e.g. in Github, Azure DevOps, BitBucket). We would caution teams to establish consensus around using a checklist *before* integrating it into the integration process.

Evaluating code review quality

It is helpful to have an understanding of your baseline quality and failure metrics before putting a checklist in place. Ultimately, a code review process should be aiming to reduce the number of defects that are released to production. It needs to balance this with the cost of the code review process itself.

Some possible measurements to consider are:

- Mean Time Between Failures (MTBF)
- Overall Equipment Effectiveness (OEE)
- First Pass Yield
- Cost of changes

Further work is needed to identify the most effective measure of success in code review quality. Your thoughts and ideas are welcomed!