All this text wasn’t written by me (Tiago a.k.a TigaxMT).

All of it was transcribed by me from HackerOne video lessons.

All credits and thanks go to them.

Source Code Review

METRICS

The traditional metric for code review is lines of code.
The number I frequently hear thrown around is 100KLOC per person-week when using static analysis tools.
I go back and forth on whether or not this is a reasonable number. Static analysis tools do cut way down on the work you need to do, but they also have many false positives.

When I'm scoping such a project and I'm not planning to rely solely on static analysis tools, I much prefer another metric: number of files.
While this can fall down in the case of massive files (and I do check for this case ahead of time), 10 files per day for manual review is absolutely reasonable and doable.

QUALITY

The quality of source review rests on whether or not you're manually reviewing the whole app. Using static analysis tools is quick, but it won't reliably find many classes of bugs. When doing manual review, you're going to work at 1% of the speed, but you're going to have a much higher quality result.

BALANCE

But all of this is a balancing act. It's simply not viable to review huge apps manually, and automated scanning is cumbersome and prone to failure.
So my recommendation is this: <100KLOC is manual review with augmentation from static analysis tools.
The opposite is true of >100KLOC, with manual review for authentication, credential storage, crypto, and other things of that nature.

COMMUNICATION

If you're doing this in a consulting environment, communication is absolutely critical with all of this, to ensure the client knows what they are going to receive.
In most cases, you will have a good idea of what you can actually accomplish given the timeframe, so just communicate that to the client.

STATIC ANALYSIS TOOLS

FORTIFY

Fortify is one of the most well-known static analysis tools. What's really interesting is that it doesn't just look at your code, but rather actually compiles it. This allows extremely deep analysis of how functions tie together and can , for instance, find SQL injection that crosses multiple function boundaries.

All of this means that it's a great product for certain cases, but its prices start at 5 figures USD and go rapidly up from there. This makes it impractical for many purposes.
In addition, it's really only terribly useful for C, C++, C# and Java.

CHECKMARX

Checkmarx is a very new tool, but in my experience it's excellent for web apps.
It's primarily useful for Java and C#, but it's a fraction of the price of Fortify and does just as good a job.

The interesting thing about this product is that it takes the entire codebase and transforms it into, essentially, a queryable abstract syntax tree. This could let you do entrypoint analysis and more, fine-tuned for your exact case.
However, they typically restrict its queries to doing vulnerability scanning and make this kind of exercise difficult. I believe there's great potential here for the future.

OTHERS

- Commercial
  - Coverity – Not as useful for web apps
  - Veracode – Hosted service, but you give them binaries rather than source (usually). Expensive, but effective
- Open Source
  - OWASP Static Analysis tools – Largely unmaintained and need a lot of work, but focused on the web.

FAILINGS

False positives are the big concern with static source analysis tools. You can get thousands of false positives from small code bases, making it easy to lose real bugs in the mix.

Most of the time, I end up writing scripts to verify the contents quickly, as the false positives in a codebase will often fall into a simple pattern.

DIVISION OF LABOR

COMPONENT-BASED SPLITTING

When dividing a large review between multiple testers, the simplest route is to divide the entire codebase into its component parts.
Each tester will take a number of components, which they test independently. Progress tracking is generally done by file counts, which can be deceptive as many files are smaller than others.

The core issue with this approach to division of labor is that many components will take less time than others, so you end up shuffling components between testers frequently.
To rectify this, communication between testers is essential. I generally recommend <10 minute daily status meetings.

CLASS-BASED SPLITTING

As an alternative to component-based splitting, splitting vulnerability reports by vulnerability class can be very effective. This is particularly useful for tests where you're reviewing a large number of findings from static analysis tools.
Each tester picks vulnerability classes that they want to review, and validates just those.

This sort of splitting makes short work of reports, as you don't have as many mental context-switches; you're always testing SQLi, XSS, etc independently.
The biggest issue here is that it's easy to miss higher-level structure issues in applications, but that plagues any review centered around static analysis reports.

FILE-BASED SPLITTING

For pure manual review by a single tester, I generally recommend splitting the project up by files and tracking progress by file counts.
This is not perfect, primarily in that it's hard to perform the exact same analysis on every file you look at, but it's the most straightforward approach.
Past a few hundred files, though, it just doesn't scale.

MEASURE, MEASURE, MEASURE

The most important thing to do when performing a source review, though, is to measure your progress frequently; at least once a day.
This will prevent you from overshooting deadlines and tell you when you have extra time to take a deeper look into certain bugs.

TIPS AND TRICKS

SOURCE MANAGEMENT

When working on a large source review, I always put the source into Git. Then I can comment on source to annotate where things could become issues, if other parts of code trigger it properly. I can then track these those as I see the code that hits the relevant APIs, and find interesting bugs.

The handy thing about having it in Git is that it's very easy for your whole team to keep track of their notes in one place, dramatically cutting down on effort and increasing bugs discovered.

USE THE PARSE

When I'm reviewing a web app codebase, the first thing I do figure out which function(s) they're using for database queries. Then I write a super using concatenated strings or formatting.
Most of the time, this will just find places where SQL queries are split across lines. But it will often find juicy SQL.

Most people shy away from writing scripts like this, but there's no better use of your time. Remember: it neither needs to work for every codebase nor needs to be perfect.
If it takes a day and cuts 3 or 4 days off your test, while giving you better results, that's a huge net win. I personally found > 300 SQLi in an app in a day using this strategy. Coverity missed them all.

Other things that I find super useful:

- Find any ouput with $_GET or $_POST
- Find all the entry-points
    - Optionally display their authorization checks

These can be done with tiny bash scripts.

ENTRYPOINT TRACING

One of the ways I like to do source review is to write a script that finds the entrypoints, then walk from there down. This has a number of benefits:

- You see code in the context of its use, not just what it's doing
- You avoid irrelevant code when all you really care about is how procedure X works

This is particularly useful when reviewing webservice code, in my experience. Here you have a small number of entrypoints generally, but often have a large amount of code on the backend.
Being able to see just what’s relevant is crucial to understanding it from a high level and finding good bugs.