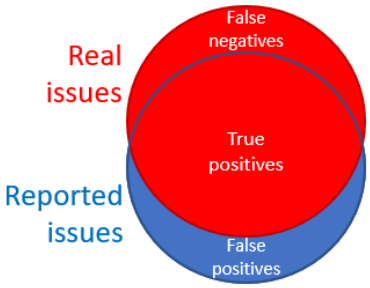
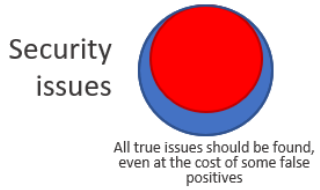
False Positives

Version 0.2

- **Static analysis** aims to detect quality issues and risks across the entire code base of an application, often to find issues more efficiently than time-consuming activities like testing and code reviews
- In static analysis, **False Positives** broadly refers to issues identified by the analysis which on inspection turn out not to be real issues
- The opposite of false positives is **False Negatives**: real issues that are *not* detected
    - False Negatives typically get less publicity but can represent a significant **risk**, especially in the area of Security
- There can be many different **reasons** for False Positives (see next slide)
    - False positives may of course be due to **errors or limitations** in the rule implementation, but often they are also due **other factors** such as knowledge about the application input or environment
- Implementing a rule with a good **balance** between false positives and false negatives is not an exact science:
    - For issues with **high impact** like security risks, or issues that are difficult to find via testing/debugging, a few hours reviewing false positives may well be worth avoiding tricky bugs in operation that may require many days of debugging
        - Example: rare memory leaks, comparing floating numbers using equal
- Seeing the forest for the trees: one false positive between 20 issues may attract most of the attention, but the most effective approach is to focus on the 19 real issues
- Impact on quality **grades**:
    - Experience shows that false positives rarely have a significant impact on the quality scores
- False Positives may arise from different situations:
    - The specific **environment** or **input** for an application may mean that certain situations will never occur in real life
        - Example: security risks may be acceptable for applications that are used by a small group of trusted users on a closed network
        - Example: division by 0 may never happen if input data ensures the divider can never be 0
    - Specific system **requirements** to an application may reduce or eliminate a risk.
        - Example: for an application which is restarted for each new set of input data and only runs for a short time, memory leaks may never build up to become an issue
    - The **non-functional requirements** to an application may influence which issues are considered ‘false positives’
        - Example: performance issues may not be relevant for code executed once a year with a full weekend available to finish processing
    - Specific libraries or frameworks may eliminate issues that the application code is not handling itself
        - Example: sanitization of SQL queries
    - The **implementation** of a specific rule may be not cover every code pattern or may contain errors
        - Example: similar syntax may have very different affects
- NB. The experience-level of the developers may also influence the amount of false positives:
    - Very experienced developers are better at writing high quality code, leading to a less issues and therefore a higher percentage of false positives than for junior developers





All true issues should be found, even at the cost of some false positives

Reduce false positives to save developers time, even at the cost of more false negatives

A download of this content is available here