96% of scanned applications contain open source components (OSC)\(^1\)

70% of the applications contain potential vulnerabilities in their OSCs on initial scans \(^2\)

---

1. Z. Zorz, "The percentage of open source code in proprietary apps is rising," 2018.
Figure 13: Survival curve of flaw closure


While many flaws are being addressed promptly, older flaws tend to linger over time.
**Independent Researchers**

- In-depth analysis from different angles.
- Focus on popular OSC. Does not leave time for us to patch.

**Company Policy**

- Analysis software that check for existing vulnerabilities.
- Limit to a list of OSCs may not be practical. What about undiscovered vulnerabilities?

**Targeted Analysis**

- Verify exploits and give appropriate response.
- Who? How? At what cost?
How should we address the potential hidden security risk of the open-source libraries?

Define framework
- Risk indicators
- Popularity ratio

Apply framework
- Risk concentration
- Methodology

Security analysis
- Key findings
The many direct and indirect indicators of security risks can be broadly categorized as

**Security Status**
- Reported vulnerabilities from CVE and NVD
- Severity of vulnerabilities based on CVSS
- Security of the language
- Typical time to remediation
- Number of open issues

**Code Characteristics**
- Lines of code
- Complexity of the code
- Number of versions
- Time of creation
- Number of libraries they use
- Where/how it is used

**Popularity**
- Number of contributors
- Number of Github stars
- Number of forks and pull requests
- Number of subscribers
- Number of downloads
- Number of dependencies

So much information...so many OSCs...
Popularity ranking of OSCs is based on the ratio of popularity for each OSC:

$\text{Relative Popularity ratio} = \frac{\text{Internal popularity}}{\text{External popularity}}$

**Internal popularity** = number of dependents

**External popularity** = average of
- Dependents
- Weekly downloads
- Github stars
Coverage for top 3 languages

Top 50 of the 1000 components:
• Covers 26.2% of dependents for C
• Covers 37.7% of dependents for Javascript
• Covers 25.9% of dependents for Java
Coverage over 3 years for JavaScript

Top 50 of the 1000 components:
- Covers 37.7% of dependents for 2019
- Covers 23.7% of dependents for 2020
- Covers 37.9% of dependents for 2021
Case Study

Focus of analysis

Language
- JavaScript

Coverage
- Top 50

Targeted analysis
- Referencing OWASP top 10 and SANS top 25
- Source code flaws
- Server side flaws
Case Study

Method

Collect internal info
- SCA
- Name and number of dependents
- Top 100 JavaScript

Collect external info
- Npm repo (weekly downloads, number of dependents)
- Github (number of stars)

Identify targets
- Calculate relative popularity of each OSCs
- Rank accordingly and cut down to top 50

Third party analysis
- Comcast Center of Excellence for Security Innovation at UConn
- Code injection

Verification
- Undiscovered
- Exploitable

3 exploitable OSC
1 deprecated OSC
Conclusion

What did we learn?

- There is a hidden risk in unpopular OSCs
- Some of them are easy to fix

Where to start?

- Take inventory of OSCs using SCA to identify existing vulnerabilities and patches
- Identify areas of concentrated risk

Security Status
- Reported vulnerabilities
- Severity of vulnerabilities
- Security of the language
- Typical time to remediation
- Number of open issues

Code Characteristics
- Lines of code
- Complexity of the code
- Number of versions
- Time of creation
- Number of libraries they use

Popularity
- Number of contributors
- Number of Github stars
- Number of subscribers
- Number of downloads
- Number of dependencies
Thank You!

Chujiao Ma
Senior Security R&D Engineer
Comcast
Chujiao_ma@comcast.com