The Devil’s in the Dependency: Data-Driven Software Composition Analysis

Ben Edwards
Cyentia Institute

Chris Eng
Veracode
We’re going to demonstrate, with data...

Even the smallest library (162 LoC) can introduce flaws into your application.

Most libraries aren’t even directly included, but are included by other libraries – a blind spot for developers.

More libraries doesn’t always mean more problems.

There are better ways to prioritize fixes than by severity.

Rejoice! 81% of patchable vulnerabilities can be fixed with a minor library update, and most updates are small – even when updates introduce new flaws!
About us

Ben

Senior Data Scientist @ Cyentia

PhD in CS applying data science to security

Wide gamut of published research: breaches, botnets, AI security, privacy, policy, and cyberwar

Chris

Chief Research Officer @ Veracode

20+ years in application security: build, break, and defend

Been involved with SoSS since Volume 1 (2010)
About the report

Veracode State of Software Security (SoSS), released annually-ish since 2010

Joint venture with Cyentia Institute since 2018

Motivations

• Insights into industry performance
• Customer benchmarking
• Actionable advice for improving AppSec

This talk includes additional research not covered in the original report!
Agenda

Data sources and biases
Library usage
Transitive dependencies
Flaw categories and patterns
Fix prioritization, evolved
Update chains
Data sources

Largest known quantitative study of application security findings

12 months of application scan data

Over 85,000 unique applications and 351,000 unique libraries
Biases

Experimental errors: Type I (false positives) and Type II (false negatives)

Selection bias, e.g. who are Veracode’s customers, which applications did they choose to analyze, etc.

Attribution bias, e.g. inclination to “blame” outcomes on things that seem relevant (e.g. developer skill) vs. other situational factors (e.g. release deadlines)
Library usage
The Open Source popularity contest
Library usage is highly language dependent

<table>
<thead>
<tr>
<th>Language</th>
<th>10th Percentile</th>
<th>Geometric Mean</th>
<th>90th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>66</td>
<td>377</td>
<td>1.4k</td>
</tr>
<tr>
<td>Multiple</td>
<td>34</td>
<td>283</td>
<td>1.4k</td>
</tr>
<tr>
<td>Ruby</td>
<td>14</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>3</td>
<td>43</td>
<td>195</td>
</tr>
<tr>
<td>PHP</td>
<td>4</td>
<td>34</td>
<td>94</td>
</tr>
<tr>
<td>.NET</td>
<td>2</td>
<td>43</td>
<td>192</td>
</tr>
<tr>
<td>Python</td>
<td>3</td>
<td>16</td>
<td>69</td>
</tr>
<tr>
<td>Go</td>
<td>2</td>
<td>13</td>
<td>49</td>
</tr>
<tr>
<td>Swift</td>
<td>1</td>
<td>4</td>
<td>22</td>
</tr>
</tbody>
</table>
Usage rate of popular libraries

Percent of applications using library
We need to talk about JavaScript...
We need to talk about JavaScript
Top 10 libraries

Incredibly numerous and small libraries
We need to talk about JavaScript Top 10 libraries

Incredibly numerous and small libraries

Top 3 have < 1 kLOC each (36, 790, 162 respectively)
We need to talk about JavaScript

Top 10 libraries

Incredibly numerous and small libraries

Top 3 have < 1 kLOC each (36, 790, 162 respectively)

isarray is only 4 lines long
We need to talk about JavaScript Top 10 libraries

Incredibly numerous and small libraries

Top 3 have < 1 kLOC each (36, 790, 162 respectively)

isarray is only 4 lines long

debug and ms have versions with CVEs
SemVer, the closest we can get to a standard...

1,300 unique 1.0.x library version strings

17.9% of libraries used have major version of 0

35,000 version strings with the format 1.11.x (representing various libraries in Amazon's Java AWS SDK)

17.4% of libraries used have a major version greater than or equal to 4
Transitive dependencies
It’s libraries all the way down...
Definition / implications

Libraries, like applications, aren’t built in a vacuum

Including a library means including every library it uses

Two types of dependencies

- **Direct Libraries** that are explicitly included by the developer
- **Transitive Libraries** that are included by another library
Transitive by language (Fig 4)

<table>
<thead>
<tr>
<th>Language</th>
<th>Apps with more direct dependencies</th>
<th>Balanced</th>
<th>Apps with more transitive dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NET</td>
<td>89.7%</td>
<td>9.3%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Swift</td>
<td>49.8%</td>
<td>43.7%</td>
<td>6.5%</td>
</tr>
<tr>
<td>Go</td>
<td>42.7%</td>
<td>23.9%</td>
<td>33.5%</td>
</tr>
<tr>
<td>Python</td>
<td>33.8%</td>
<td>24.3%</td>
<td>41.9%</td>
</tr>
<tr>
<td>Java</td>
<td>34.9%</td>
<td>7.4%</td>
<td>57.7%</td>
</tr>
<tr>
<td>PHP</td>
<td>6.5%</td>
<td>12.6%</td>
<td>80.9%</td>
</tr>
<tr>
<td>Ruby</td>
<td>2.2%</td>
<td>16.1%</td>
<td>81.7%</td>
</tr>
<tr>
<td>JavaScript</td>
<td>10.2%</td>
<td>2.9%</td>
<td>86.9%</td>
</tr>
</tbody>
</table>

Percent of applications
Direct vs Transitive vulnerabilities (Figs 11-12)

- 70.5% with library flaws
- 29.5% without library flaws
- 46.6% transitive
- 41.9% direct
- 11.5% both
Flaws

Every rose has its thorn...
More libraries = more problems? (Fig 13)
Language choice makes a difference (Fig 5)
OWASP Top Ten (Fig 6)

<table>
<thead>
<tr>
<th>Issue</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7-Cross-Site Scripting (XSS)</td>
<td>29.1%</td>
</tr>
<tr>
<td>A8-Insecure Deserialization</td>
<td>23.5%</td>
</tr>
<tr>
<td>A5-Broken Access Control</td>
<td>20.3%</td>
</tr>
<tr>
<td>A1-Injection</td>
<td>8.8%</td>
</tr>
<tr>
<td>A3-Sensitive Data Exposure</td>
<td>7.8%</td>
</tr>
<tr>
<td>A2-Broken Authentication</td>
<td>7.4%</td>
</tr>
<tr>
<td>A4-XML External Entities (XXE)</td>
<td>2.7%</td>
</tr>
<tr>
<td>A6-Security Misconfiguration</td>
<td>0.6%</td>
</tr>
</tbody>
</table>
### PHP is basically a minefield (Fig 7)

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Go</th>
<th>Java</th>
<th>JavaScript</th>
<th>.NET</th>
<th>PHP</th>
<th>Python</th>
<th>Ruby</th>
<th>Swift</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1-Injection</td>
<td>3.4%</td>
<td>1.7%</td>
<td>2.5%</td>
<td>2.9%</td>
<td>18.6%</td>
<td>6.3%</td>
<td>7.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td>A2-Broken Authentication</td>
<td>4.9%</td>
<td>6.9%</td>
<td>1.9%</td>
<td>1.9%</td>
<td>21.3%</td>
<td>6.5%</td>
<td>3.2%</td>
<td>0.2%</td>
</tr>
<tr>
<td>A3-Sensitive Data Exposure</td>
<td>8.0%</td>
<td>2.1%</td>
<td>0.6%</td>
<td>8.8%</td>
<td>4.6%</td>
<td>2.6%</td>
<td>1.4%</td>
<td>6.1%</td>
</tr>
<tr>
<td>A4-XML External Entities (XXE)</td>
<td>0.0%</td>
<td>5.9%</td>
<td>0.0%</td>
<td>0.5%</td>
<td>0.1%</td>
<td>1.6%</td>
<td>0.5%</td>
<td>0.2%</td>
</tr>
<tr>
<td>A5-Broken Access Control</td>
<td>10.7%</td>
<td>8.9%</td>
<td>4.9%</td>
<td>14.8%</td>
<td>22.5%</td>
<td>9.4%</td>
<td>8.0%</td>
<td>7.7%</td>
</tr>
<tr>
<td>A6-Security Misconfiguration</td>
<td>0.0%</td>
<td>0.7%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>1.2%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>A7-Cross-Site Scripting (XSS)</td>
<td>11.0%</td>
<td>10.5%</td>
<td>11.6%</td>
<td>8.4%</td>
<td>40.1%</td>
<td>13.3%</td>
<td>13.9%</td>
<td>0.0%</td>
</tr>
<tr>
<td>A8-Insecure Deserialization</td>
<td>0.0%</td>
<td>7.6%</td>
<td>0.0%</td>
<td>0.4%</td>
<td>17.4%</td>
<td>0.9%</td>
<td>1.5%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>
Alright, now what?
Prioritizing fixes
Not all vulnerabilities have exploits (Fig 8)

- PHP: 27.1%
- Java: 15.7%
- .NET: 14.2%
- Go: 13.9%
- Ruby: 12.3%
- Swift: 11.3%
- Python: 11.3%
- JavaScript: 6.5%

20.7% of all vulnerable libraries have a PoC exploit.
PoC exploits by OWASP category (Fig 10)
The vulnerability funnel (Fig 14)
Good news: most fixes are minor (Figs 16-17)

Out of that 73.8%...

- 73.8% can be fixed with update
- 26.2% no update available

- 38.1% minor version
- 40.9% patch
- 18.7% major version
- 2.2% revision
It’s never that easy...

Updating a library can introduce new flaws, which require further updates, which may introduce new flaws, requiring more updates...

So what do these update chains look like?
It’s never that easy...

Updating a library can introduce new flaws, which require further updates, which may introduce new flaws, requiring more updates...
So what do these update chains look like?

- Single step to version with no known flaw
- Multiple Steps to version with no known flaw
It’s never that easy...

Updating a library can introduce new flaws, which require further updates, which may introduce new flaws, requiring more updates...

So what do these update chains look like?

- Single step to version with no known flaw
- Multiple Steps to version with no known flaw
- No update available
- Multiple Steps to version with flaws and no update available
It’s never that easy...

Updating a library can introduce new flaws, which require further updates, which may introduce new flaws, requiring more updates...
So what do these update chains look like?

- Single step to version with no known flaw
- Multiple Steps to version with no known flaw
- No update available
- Multiple Steps to version with flaws and no update available
- Suggested updates are circular
Begs many questions

How do these chains end?

How many steps do they have?

Do they significantly increase update size?
How do the chains end?

Percent of flaws in applications:

- 33.9%
- 33.1%
- 26.2%
- 6.4%
- 0.5%
Most chains are relatively short...

- Final version has no known flaws
- Final version has flaws

90.1% of flaws have chain lengths less than or equal to 4
... but it varies by language
Most updates are still small
Conclusions / Q&A
Takeaways

Open source software has a surprising, and surprisingly variable, number and type of software flaws.

The attack surface of many applications — due to the transitive dependency phenomenon — is much larger than developers may expect.

Language selection does make a difference — both in terms of the size of the ecosystem and in the prevalence of flaws in those ecosystems.

Most fixes are relatively minor in nature, suggesting that this problem is one of discovery and tracking, not huge refactoring of code.
Questions?

ben@cyentia.com

ceng@veracode.com