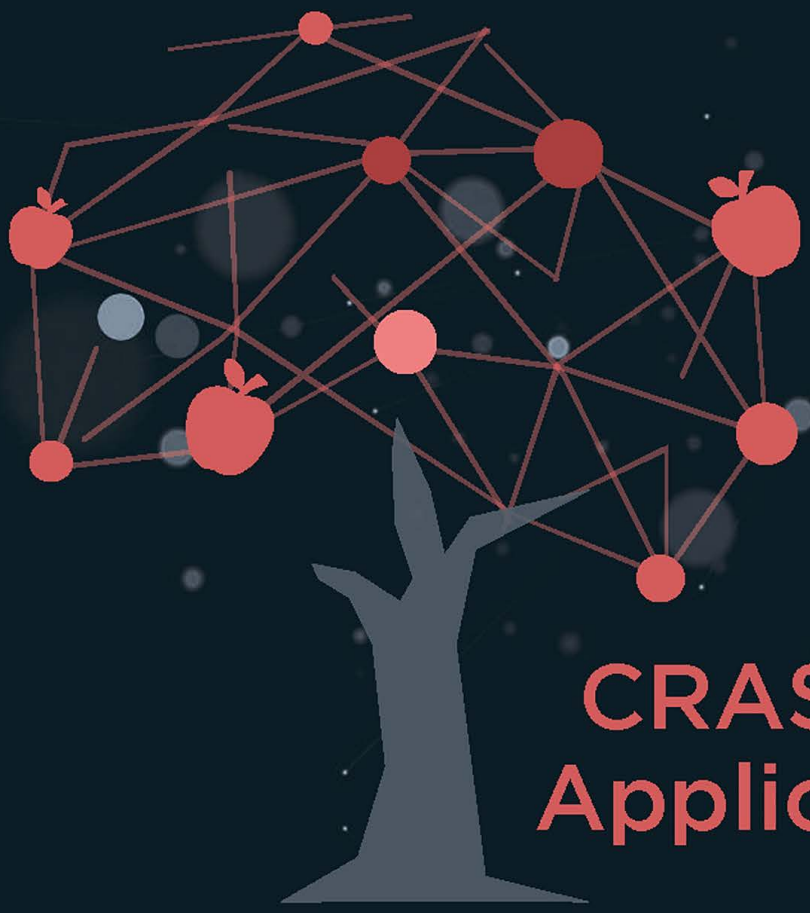
CAST
Software Intelligence for Digital Leaders

CAST RESEARCH LABS



CRASH Report on Application Security

A cross-industry benchmark on application security vulnerabilities based on Common Weakness Enumeration (CWE) analysis.

Don't let one bad apple spoil the bunch

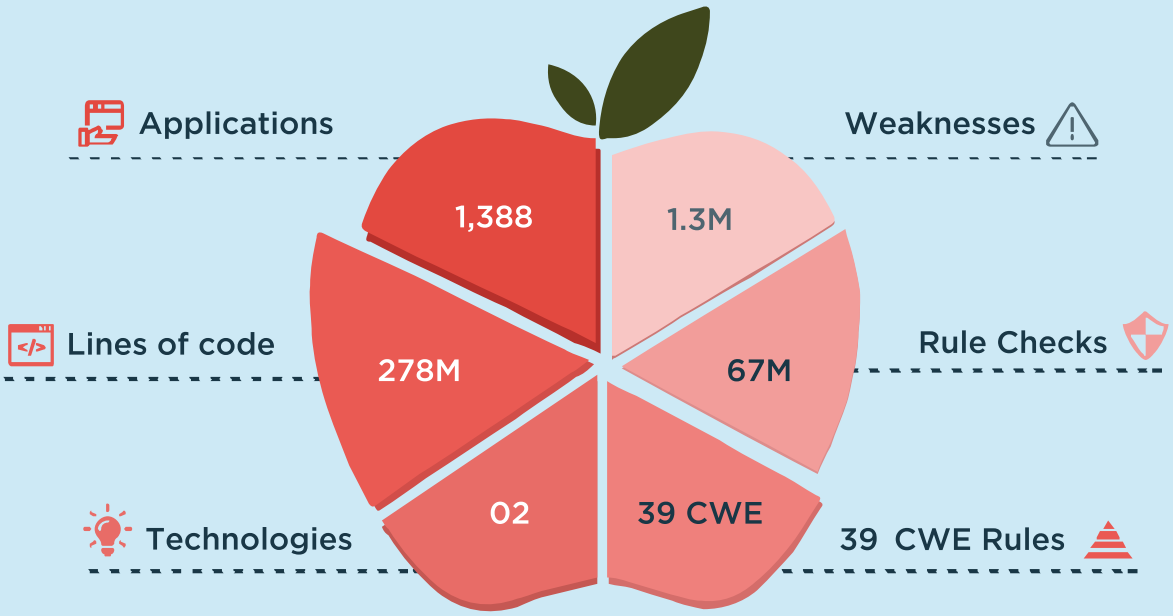
What is CRASH?

CAST Research on Application Software Health (CRASH)

What is the Common Weakness Enumeration from MITRE?

The CWE is a repository of known security weaknesses in software architecture, design or code. CWE serves as a standard way to measure an organization's defense against these common security weaknesses while providing a baseline standard for weakness identification, mitigation and prevention efforts.

The CRASH Report Sample



Definitions

Opportunity :
An opportunity to make a coding mistake or introduce a vulnerability through non-secure coding practices.

CWE :
A violation caused by a coding mistake or non-secure coding practices.

KLOC :
One thousand lines of code.

The CRASH Report Sample Distribution by Size

J-EE - 957 apps / .NET - 431 apps

Application Size Does Not Affect Opportunity or CWE Density

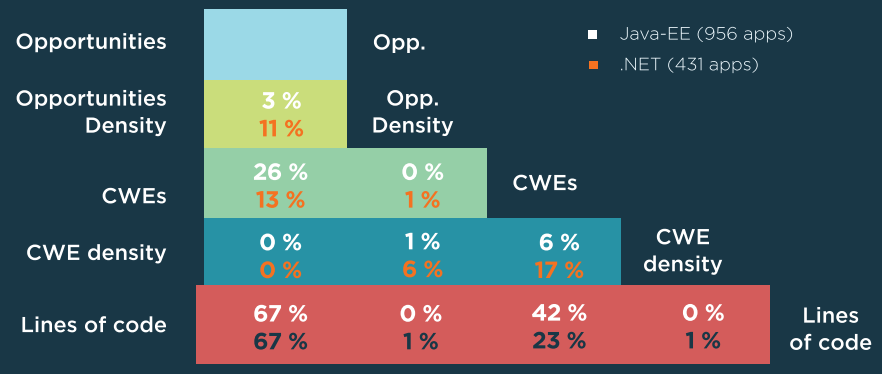
Unsurprisingly, the greater the number of opportunities for violating a CWE rule in an application, the more CWE weaknesses that occur. Both of these highly correlate with an application's size (total lines of code).

However, the density of CWE weaknesses and opportunities are not correlated with an application's size, except for a small relationship with opportunity density in .NET.

Although the size of an application affects the total number of CWEs, the density of CWE weaknesses is driven by other factors.

Probability the relationship occurred by chance:

1 in 20, ** 1 in 100, * 1 in 1000**

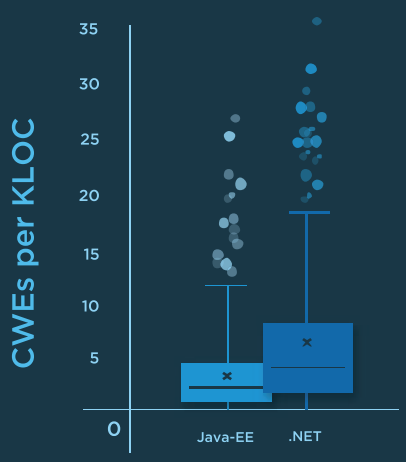


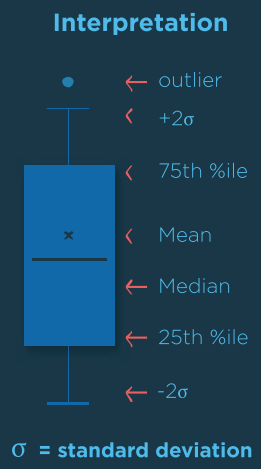
"It's not the size of the apple that matters."

The Density of CWE Weaknesses Varies by Language

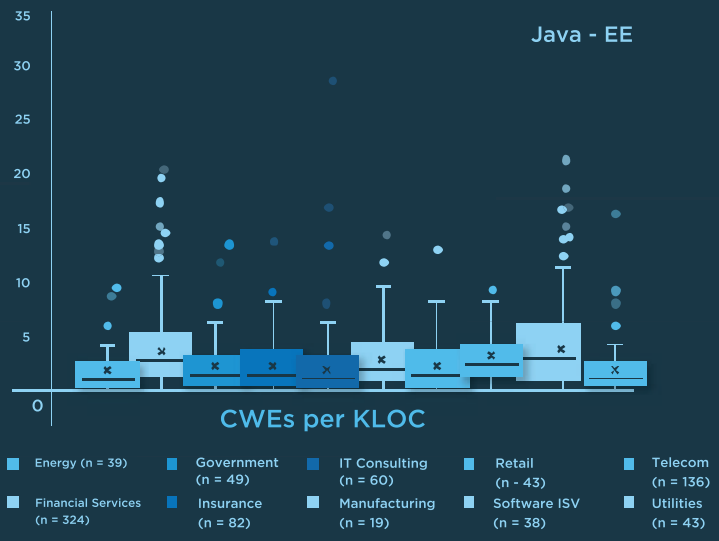
Although the distributions of opportunities for CWEs across the wide range of application sizes were similar for both Java-EE and .NET, the distributions of CWE weaknesses were significantly different (< 1 in 1000 by chance). .NET had a higher mean density of CWE weaknesses, as well as greater variance in CWE density scores and a wider range.
Since the medians were similar, the difference in means indicates a far greater range of CWE weaknesses in .NET, some having a density of greater than 35 weaknesses per KLOC.

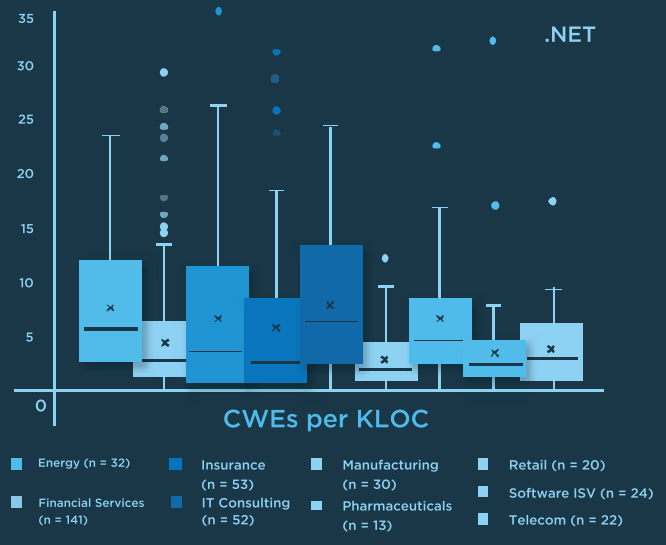
"Some apples fall way too far from the tree."





Application Security by Industry Segment





Financial Services, Telecom, and IT Consulting had the highest mean CWE densities. Energy and Utilities had the lowest CWE densities as well as the least variation in CWE density scores. The differences between the means of the industries with the lowest and highest CWE density scores is almost a factor of 2, as are differences in the sizes of their interquartile ranges (25th to 75th percentile scores). While all industry segments had mean CWE density scores below 5 CWEs per KLOC, all but Energy had applications containing more than 10 CWEs per KLOC.

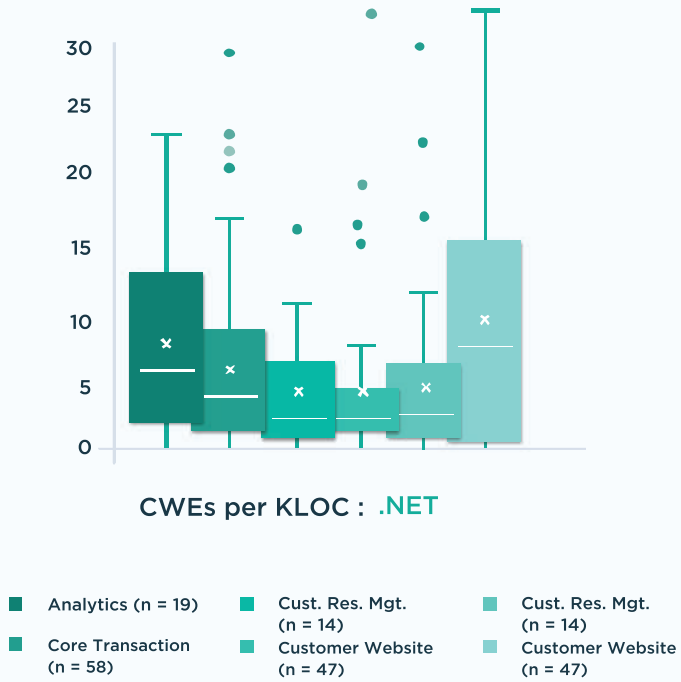
The pattern in .NET applications was different than in Java-EE. Mean CWE density s cores were almost twice as high in Energy, Insurance, IT Consulting, and Manufacturing compared to their scores for Java-EE applications. In most industry segments, variation in CWE densities was much larger than in .NET. While most applications across industry have less than 5 CWEs per KLOC, there are many applications well above this density, ranging into the 20s and even 30s per KLOC, presenting serious security risks.

Application Security Differed Among Application Types in .NET

Mean CWE density scores among types of applications were not significantly different in Java-EE. However in .NET significant differences among the application types accounted for 5% of the variation in density scores.

ERP and Analytics (too few apps) had the highest CWE densities per KLOC among the application types. The variation in density scores was also much larger in these two application types.'

Customer Website, Customer Resource Management (too few apps), and Enterprise Portals had the lowest CWE density scores. However, they still had some high density outliers.

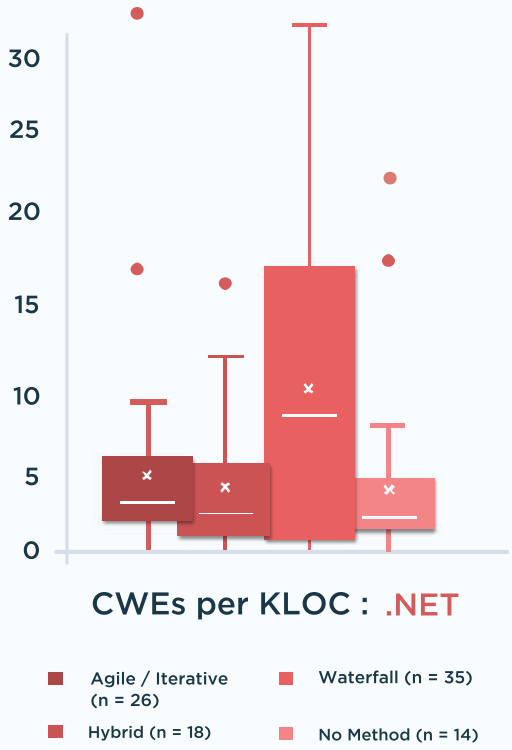


Application Security by Development Methodology - In .NET, the Worst Apples Are Found at the Bottom of the Waterfall

CWE density across Java-EE applications was fairly consistent for applications developed with agile/iterative, hybrid (agile/waterfall mix), or waterfall methods. Although applications developed without a method appeared to have higher CWE densities, there were not enough 'no method' applications for these differences to be statistically significant.

NET, however, had statistically different results even though the samples sizes were smaller than desired.

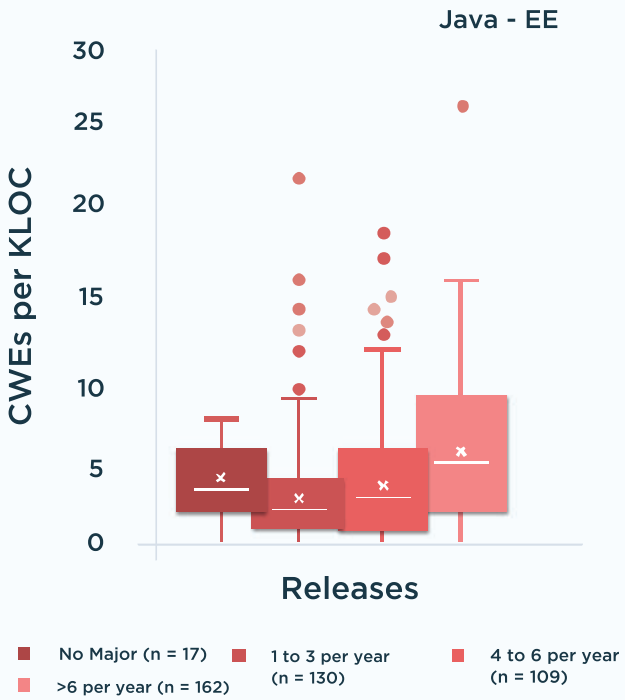
While applications developed with Agile, Hybrid, or No Method were quite similar in CWE density, the density of security weaknesses exploded in those developed with Waterfall methods. In fact, 75% of CWE densities in applications developed with other methods would fall in the lower half of those developed through a waterfall. Thus .NET applications developed in a waterfall are particularly prone to cyber-attacks and exploit.



The More You Release, the More You Contaminate the Crop

Java-EE applications released more than 6 times per year had significantly higher CWE densities than those released less than 6 times per year. This difference accounted for 8% of the variation in CWE density scores. This difference was not found among .NET applications.

This is a particularly interesting finding, given the industry's broad shift toward Agile development and continuous release schedules. While this kind of deployment cycle has been shown to improve user experience, it is putting the application at greater risk of security defects which could ultimately cause an excruciating user experience.

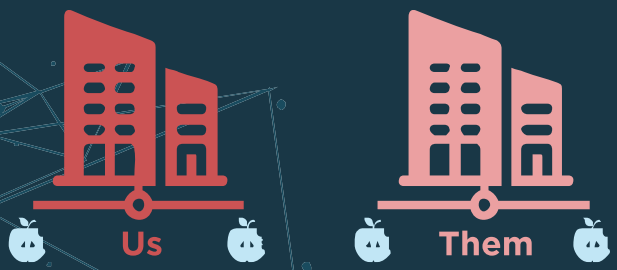


"An apple a day keeps the doctor away, unless you eat bad ones frequently"

What Didn't Matter

Source

For both Java and .NET, there were no significant differences in CWE density between applications developed in-house versus those that were outsourced.



Shore

For both Java and .NET, there were no significant differences in CWE density between applications developed onshore versus those developed offshore.

" Bad apples can be grown anywhere, but they don't seem to cluster in one place "

Summary of Factors Impacting Application Security

- Evaluate application security scores across the portfolio. CWE densities are not related to application size. Large or small, some applications are just more insecure.
- Financial Services and Telecom had the highest CWE densities in .NET while a different combination of industries had higher CWE densities in .NET. However, differences among industry segments were not as large as those caused by other factors.
- Java applications developed with Waterfall methods had the worst CWE densities and large variation in scores.
- Watch out for particularly sour apples in .NET applications. On average, these apps had higher CWE densities and greater variation in scores than those developed in Java.
- Neither the source (inhouse vs. outsourced) nor the shore make a difference in CWE densities.
- Rethink your Agile continuous release cycles. Java apps released more than 6 times per year had the highest CWE densities.

Factor	Statistically Significant ?		% of variation accounted for	
Language	YES		7%	
	J-EE	.NET	J.EE	.NET
Industry	YES	YES	4%	5%
App Type	NO	YES		6%
Source	NO	NO		
Shore	NO	NO		
Method	NO	YES		12%
Releases	YES	NO	8%	

"Don't upset the applecart, disciplined practices have the strongest effect on application security"

CWEs analyzed in this report

.NET: Close SQL connection ASAP

Avoid white-listing the "dirname" attribute in user generated content

Implement success and error callbacks when using .ajax interface.

The exception Exception should never been thrown. Always Subclass Exception and throw the Subclassed Classes (CWE-396)

Avoid catching an exception of type Exception, RuntimeException, or Throwable (CWE-397)

Avoid using oninput in body containing input autofocus

Avoid using onscroll event with autofocus input

Avoid using Process.exit()

Avoid using setData in ondragstart with attribute draggable set to true

Avoid using source tag in video/audio with event handler

Avoid using submitted markup containing "form" and "formation" attributes

Avoid using video poster attributes in combination with javascript

Avoid uncontrolled format string (CWE-134)

Avoid using autofocus and onblur in submitted markup

Avoid using autofocus and onfocus in submitted markup

Avoid using import with external URI

Avoid "id" attributes for forms as well as submit

Avoid improper processing of the execution status of data handling operations

Avoid Log forging vulnerabilities(CWE-117)

Avoid numerical data corruption during incompatible mutation

Avoid the lack of error handling in the Node.js callbacks

Avoid the use of JSON.parse and JSON.stringify in AngularJS web app

J2EE:Never miss dataflow proven cross-site scripting injection flaws(CWE-79)

J2EE:Never miss dataflow proven file path manipulation flaws (CWE-73)

J2EE:Never miss dataflow proven LADP injection flaws (CWE-90)

J2EE:Never miss dataflow proven OS command injection flaws (CWE-78)

J2EE:Never miss dataflow proven SQL injection flaws (CWE-89)

J2EE:Never miss dataflow proven XPath injection flaws (CWE-91)

Avoid using Javascript or expression in the CSS file

Avoid using Javascript Regexp typecheck in AngularJS application (CWE-704)

Avoid using Javascript string typecheck in AngularJS application (CWE-704)

Avoid the use of the default JavaScript implementation [].forEach in AngularJS web app

Avoid using Javascript Array typecheck in AngularJS application (CWE-704)

Avoid using Javascript Date typecheck in AngularJs application (CWE-704)

Avoid using Javascript Function typechecks in AngularJS application (CWE-704)

Avoid using Javascript Number typecheck in AngularJS application (CWE-704)

Avoid using Javascript Object typecheck in AngularJS application (CWE-704)

About the CRASH Report

APPMARQ houses data collected during system-level structural analyses of large IT applications. Structural quality refers to the engineering soundness of the architecture and coding of an application, rather than to the correctness with which it implements the customer's functional requirements. Structural quality is occasionally referred to as nonfunctional, technical, or internal quality.

CAST RESEARCH LABS conducts advanced empirical research on software-intensive IT systems. CRL provides practical advice and periodic benchmarks to the global application development community, as well as interacting with the academic community.

CAST is the software intelligence category leader. CAST technology can see inside custom applications with MRI-like precision, automatically generating intelligence about their inner workings - composition, architecture, transaction flows, cloud readiness, structural flaws, legal and security risks. It's becoming essential for faster modernization for cloud, raising the speed and efficiency of Software Engineering, better open source risk control, and accurate technical due diligence. CAST operates globally with offices in North America, Europe, India, China. Visit www.castsoftware.com.