How do JavaScript frameworks impact the security of applications

Ksenia Peguero

whoami

- *Current:* Sr. Research Engineer at Synopsys, part of the Security Research Lab
- *Prior:* Principle Consultant at Cigital/Synopsys
- PhD candidate at George Washington University
- Mother
- Ballroom dancer
- @KseniaDmitrieva

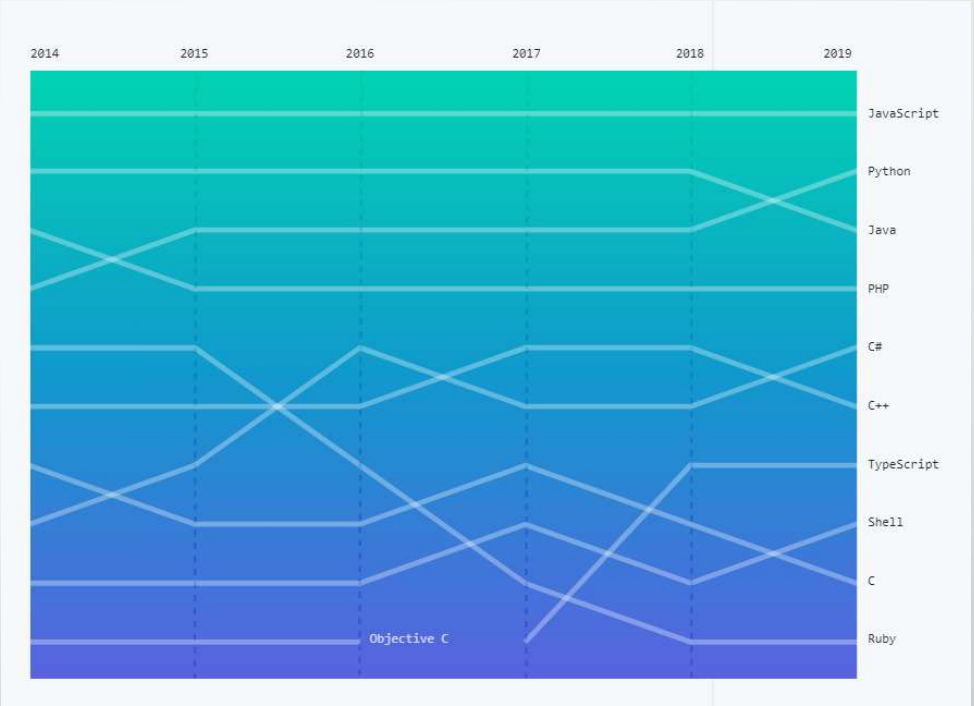
SYNOPSYS®

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Why JavaScipt?

Language popularity by open pull request according the Octoverse report from 2014 to 2019:

- JavaScript has been the leading programming language for the last 6 years
- JavaScript is used for web applications on client-side and server-side, in mobile applications, desktop applications and IoT software.

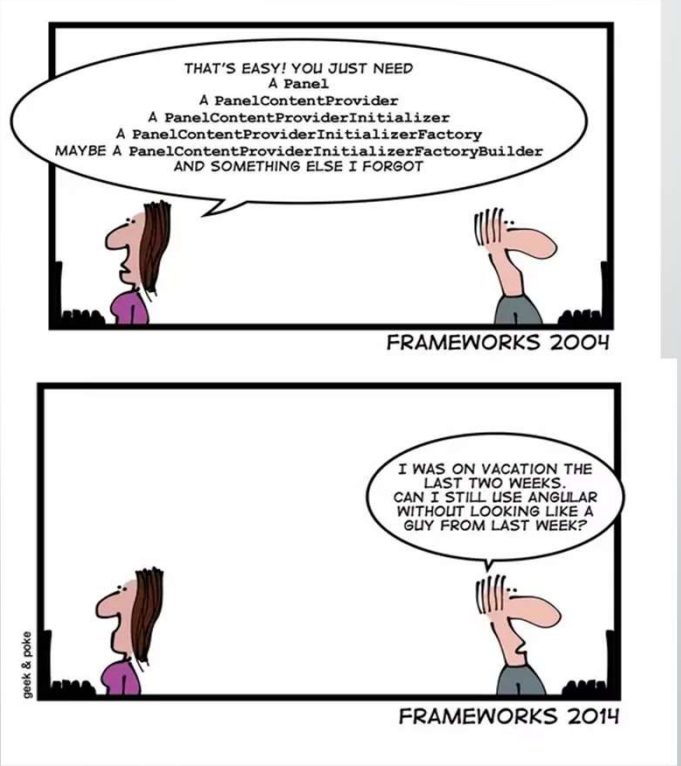


https://octoverse.github.com/

State of the Client-Side JavaScript Field Today

How many frameworks are there?

- Client-side: over 50 frameworks, according to the https://jsreport.io/
  - Angular, React, Vue
- Server-side: over 40 frameworks, according to http://nodeframework.com/
  - Express, Koa, Sails
- Full-stack frameworks
  - Meteor, Aurelia, Derby, MEAN.js
- Desktop frameworks
  - Electron
- Mobile frameworks
  - Phonegap, Cordova



What is there in the framework for security?

- Frameworks provide functionality, easiness of prototyping and development, performance…
  Hm,… security, anyone?

- Following the "shift-left" paradigm in software security, we should not only identify and fix vulnerabilities earlier in the software development lifecycle, but also **prevent** them earlier.

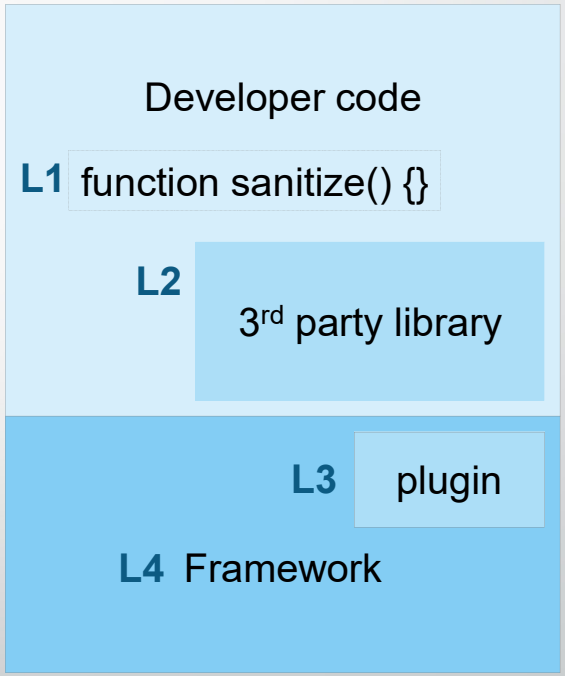
- Questions:
  - Does the security of a framework help to make applications more secure?
  - Does building security controls into a framework result in "shifting-left" the security of the application?

Levels of Vulnerability Mitigation

proposed by John Steven

A vulnerability may be mitigated at the following levels in relation to the framework:

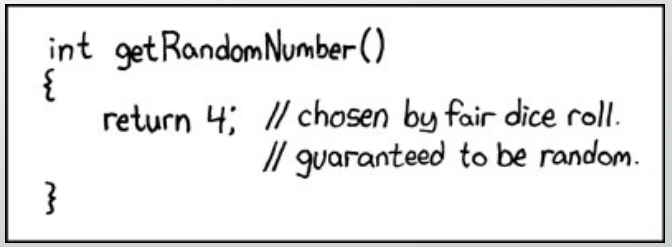
- **L0 - No mitigation in place.** Baseline – no protection
- **L1 - Custom function.** A security control written by developers
- **L2 - An external library** that provides a security control
- **L3 - A framework plugin.** A third-party code used by developers which tightly integrates with the framework
- **L4 - Built-in mitigation control** implemented in the framework as a function or feature



Mitigation Examples

- **L1 - Custom function:** developer implementation
- **L2 - An external library:** ESAPI (The OWASP Enterprise Security API) - a security control library https://github.com/ESAPI/esapi-java-legacy
- **L3 - A framework plugin:** the `csurf` plugin for Express https://www.npmjs.com/package/csurf
- **L4 - Built-in mitigation control:** Spring Security https://spring.io/projects/spring-security

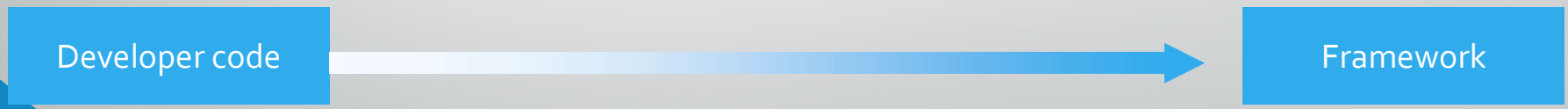
```
function cors (res) {
  res.set({
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Headers': 'Origin, X-Requested-With,
Content-Type, Accept'
  })
  return res
}
```



https://xkcd.com/221/

Hypothesis

The closer the mitigation is located to the framework itself, the fewer vulnerabilities the code will have.



Case Study 1: XSS

- **XSS** is "a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites." (OWASP)
- **Common protections**:
  - Output encoding
  - Input validation
  - Sanitization
- **Special case**: need to allow users to use some HTML, but not malicious JavaScript, for example, blog posts, marketing letters, CMS.
- **How to implement**: display raw HTML and let the browser render it.
- **How to protect from XSS**: only allow a safe subset of HTML (sanitizing the "bad" HTML or using alternative markup languages like Markdown)

Data Selection for XSS (2016)

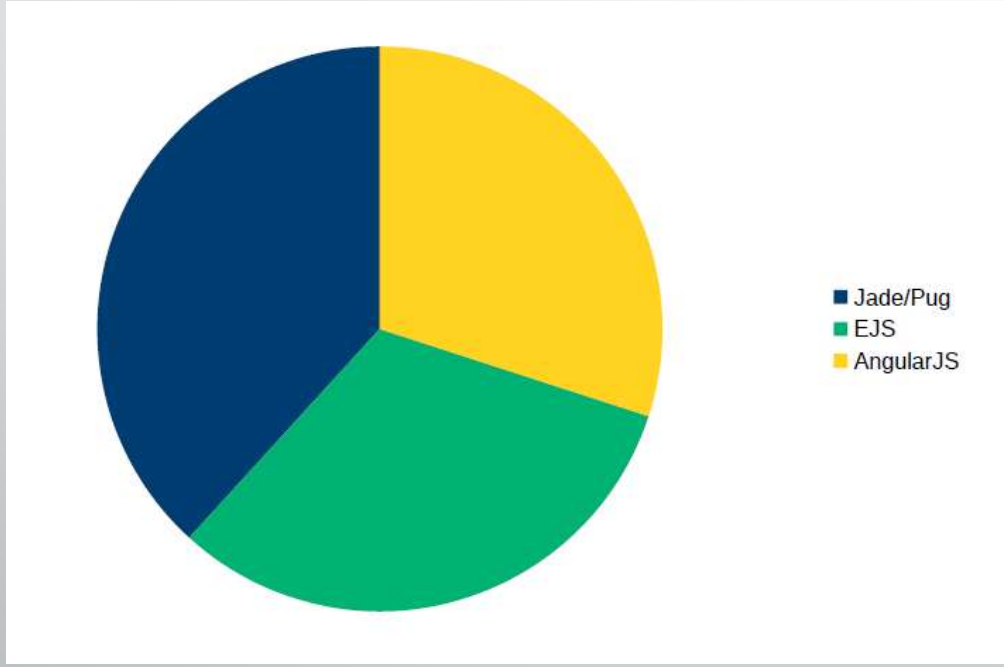
- Use case: the application needs to display user input that contains HTML markup

- **Application Selection Criteria:**
  - Application type: blog or CMS
  - Full-stack JavaScript applications
  - Template engines: Jade/Pug, EJS, AngularJS

- **Filters:**
  - last commit no later than 2013
  - at least 1 star
  - the language is JavaScript, HTML, CSS

https://insights.bookbub.com/creative-blog-post-ideas-authors/

Resulting Dataset



Total of 170 projects:

- 65 Jade/Pug
- 54 EJS
- 51 AngularJS

Escaping and Interpolation in Frameworks

- **Jade/Pug:**
  - Escaping: curly braces, equals sign for tags
  - Interpolation: bang-sign, no sanitization

```
h1=title
p {article.name}
p!=article.content
div !{post.body}
```

- **EJS:**
  - Escaping: special braces '<%=' and '%>'
  - Interpolation: '<%-' and '%>', no sanitization

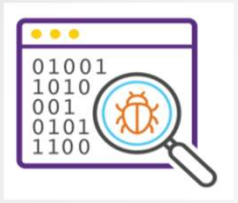
```
<%=article.title%>
<%-article.body%>
```

- **AngularJS:**
  - Escaping: contextually-aware escaping with double curly braces
  - Interpolation: safe subset with 'ng-bind-html', trustAsHtml() for raw HTML interpolation

```
<a href="{{post.url}}">
{{post.title}} </a>
<p ng-bind-html=
"post.description"></p>
```

Analysis Pipeline

Download project info and template files from GitHub



Run parser and analyzer for each template engine

Jade/Pug
Extended pug-lexer and pug-parser

EJS
Extended EJS core project, custom analyzer

AngularJS
ESLint with a custom rule

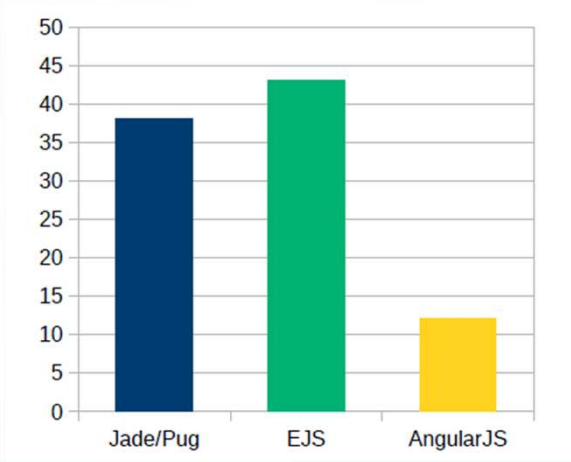
Perform manual review

Perform statistical analysis of the results

Results

Template engine	Number of projects	Number of vulnerabilities	Number of vulnerable projects	% of vulnerable projects	Mitigation level
Jade/Pug	65	72	25	38%	L1 or L2
EJS	54	96	23	43%	L1 or L2
AngularJS	51	12	6	12%	L4

Percentage of applications vulnerable to XSS



Mitigation Levels:

L1 - Custom function

L2 - An external library

L3 - A framework plugin

L4 - Built-in mitigation control

Confounding Variables Analysis for XSS

- What if AngularJS developers are just better / smarter / more experienced than developers writing Jade/Pug and EJS?
- We use ANOVA statistical analysis to verify our results against other factors

Criteria	P-value
Developer's overall experience	0.319279
Developer's JavaScript experience	0.132049
Project size	0.431335
Project popularity (stars)	0.200649
Project reuse (forks)	0.211615
Template engine	0.001021

← value < 0.05

The statistically significant difference is shown by a p-value < 0.05. The choice of a template engine is **the only factor** affecting the number of vulnerabilities.

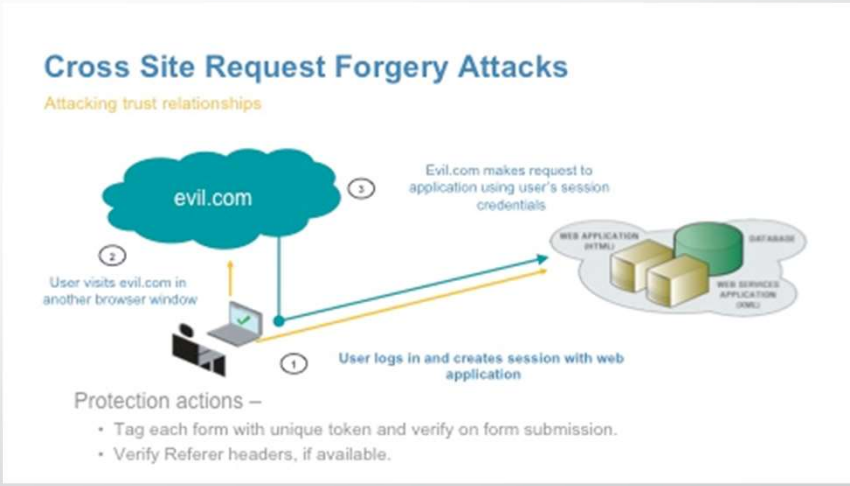
Hypothesis proved (for XSS): *the closer the mitigation is located to the framework itself, the fewer vulnerabilities the code will have*

Case Study 2: CSRF

CSRF - "an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated" (OWASP)

Protection methods:

- Server-Side:
  - CSRF tokens
    - In POST parameters
    - Double-submit cookie
  - Two-factor authentication
- Not using session cookies:
  - JWT
  - Using web socket session
- Client-side:
  - Same-site cookies
  - White-listing expected origins
  - Allowed referrer lists



https://linuxsecurityblog.com/2016/02/11/defending-against-csrf-attacks/

Data Selection for CSRF (2018)

Use case: authenticated users call sensitive functionality that change the server state

Application Selection Criteria:

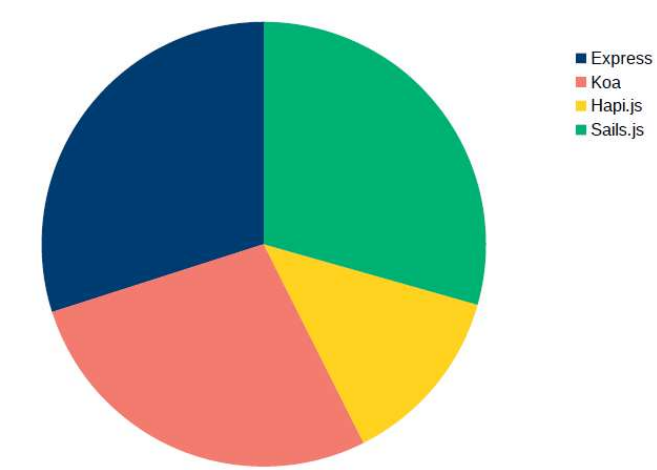
- Application type:
  - Blog
  - CMS
  - E-commerce
  - REST API
- JavaScript server-side applications
- Frameworks: Express, Koa, Hapi, Sails, Meteor*

Selection goal:

- 100 applications per framework

http://leanport.com/effective-ways-to-improve-e-commerce-marketing/

Resulting Dataset



Total of 364 projects

Framework	Blog	CMS	E-commerce	REST API	Total
Express	29	35	45	0	109
Koa	68	26	6	0	100
Hapi	26	3	9	10	48
Sails	72	20	15	0	107

CSRF Protection in Frameworks

Express:

- Plugins csurf — L3

Koa:

- Plugin koa-csrf — L3

Hapi:

- Plugin crumb — L3

Sails:

- Framework configuration — L4

Meteor:

- Framework architecture — L?

Express

```
const express = require('express');
const csrf = require('csurf');
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser());
app.use(csrf({cookie: true}));
```

Koa

```
const Koa = require('koa');
const session = require('koa-session');
const CSRF = require('koa-csrf');
const app = new Koa();

app.use(session(app));
app.use(new CSRF());
```

CSRF Protection in Frameworks

Express:

- Plugins csurf — L3

Koa:

- Plugin koa-csrf — L3

Hapi:

- Plugin crumb — L3

Sails:

- Framework configuration — L4

Meteor:

- Framework architecture — L?

Hapi

```
const Hapi = require('hapi');
const Crumb = require('crumb');

const server = new Hapi.Server({port: 8000});
(async () => {
  await server.register({
    plugin: Crumb,
    options: {restful: true}
  });
  ...
```

Sails

```
module.exports.csrf = {
   csrf.grantTokenViaAjax: true,
   csrf.origin: 'example.com'
}
```

Special Case: Meteor and JWT

A CSRF attack depends on a session being maintained in a cookie. → If there is no cookie, the attack is not possible.

Meteor:

- Meteor uses custom Distributed Data Protocol (DDP) for client-server communication
- DDP runs on WebSockets instead of HTTP
- A session is maintained via a long-lived WebSocket connection
- A third party cannot send a forged request over an established WebSocket connection

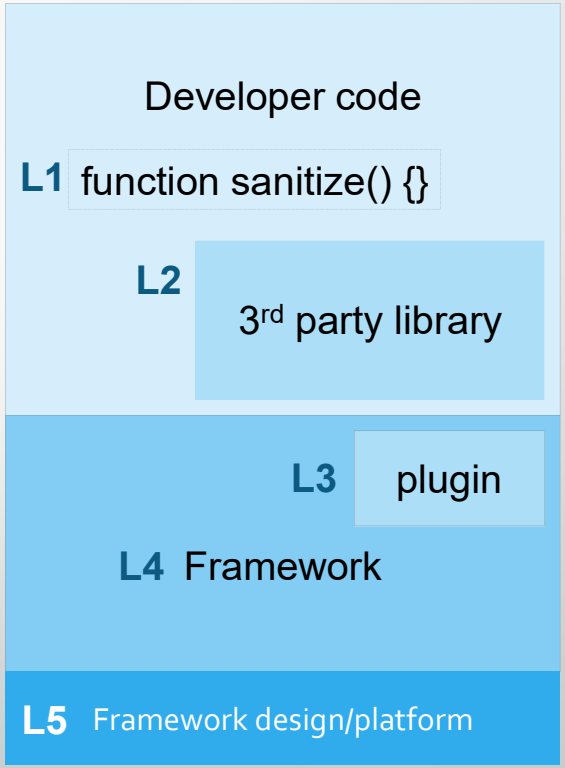
JSON Web Token (JWT):

- Developed as access tokens, but used as session tokens
- Not stored in cookies, but transmitted in HTTP headers, which are not added to cross-origin requests by the browser
- Have other limitations, but do protect from CSRF

Levels of Vulnerability Mitigation

A vulnerability may be mitigated at the following levels in relation to the framework:

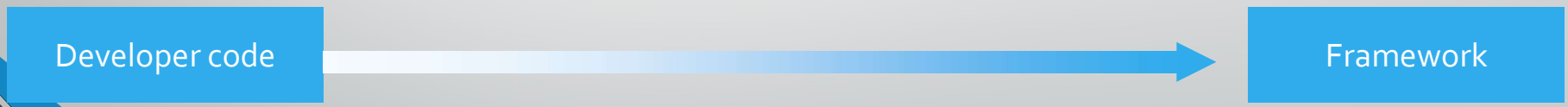
- **L0 - No mitigation in place.** Baseline – no protection
- **L1 - Custom function.** A security control written by developers
- **L2 - An external library** that provides a security control
- **L3 - A framework plugin.** A third-party code used by developers which tightly integrates with the framework
- **L4 - Built-in mitigation control** implemented in the framework as a function or feature
- **L5 – Architecture level mitigation control.** A framework is designed in a way that makes the attack impossible.



Hypothesis

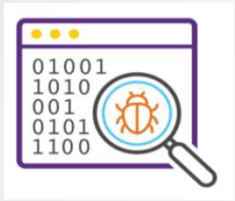
The closer the mitigation is located to the framework itself, the fewer vulnerabilities the code will have.

Does it work for CSRF?



Analysis Pipeline

Download project from GitHub



Run ESLint with custom rules

- CSRF plugins: csurf, csrf, alt-XSRF, koa-csrf, crumb, lusca
- Sails configuration
- JWT plugins

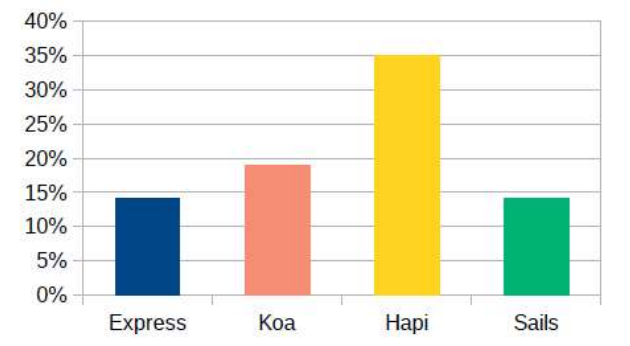
Perform manual review

Perform statistical analysis of the results

Resulting Dataset

Framework	Number of projects	CSRF protection	JWT	Total protected	% of protected projects	Mitigation level
Express	109	6	9	15	14%	L3
Koa	100	6	14	19*	19%	L3
Hapi	48	0	17	17	35%	L3
Sails	107	7	8	15	14%	L4

Percentage of applications protected from CSRF



Mitigation Levels:

L1 - Custom function

L2 - An external library

L3 - A framework plugin

L4 - Built-in mitigation control

Confounding Variables Analysis for CSRF

Results of confounding variables analysis using ANOVA statistical tests

Criteria	P-value
Developer's overall experience	0.165714
Developer's JavaScript experience	0.161450
Project size	0.263872
Project popularity (stars)	0.411852
Project reuse (forks)	0.513946
Framework	0.507734

The statistically significant difference is shown by a p-value < 0.05. However, none of the calculated p-values are smaller than 0.05. Thus, none of the confounding variables show a correlation.

For CSRF, **the hypothesis is not proved.** There is no correlation between the level of CSRF mitigation and the presence of the CSRF of vulnerability in the application, except for L5 (Meteor).

Comparing XSS and CSRF Results

- Compare %-age of protected projects by mitigation level/framework:

Why?

- L4 protection in AngularJS is enabled by default
- L4 protection in Sails is disabled by default

➢ Secure defaults are as important as the implementation levels of security controls

Conclusion

Recommendations to the framework developers and maintainers:

- ✓ Implementing security controls as third-party plugins does not ensure secure applications
  - Other solutions:
    - Processes: secure coding guidelines, training
    - Secure SDLC: secure code review, linting rules on every build, static analysis, penetration testing
- ✓ Instead, build security controls into the framework
- ✓ Ensure that the default settings are secure and that the security control is enabled
- ✓ When plausible, design a framework in a way that eliminates the possibility of the vulnerability at the architecture level

Ksenia Peguero
ksenia@synopsys.com
Twitter: @KseniaDmitrieva
https://www.synopsys.com/software