SYNOPSYS®
Silicon to Software™

Impact of Frameworks on Security of JavaScript Applications

Ksenia Peguero

whoami

- *Current:* Sr. Manager of Research Engineering at Synopsys Software Integrity Group

- *Prior:* Principle Consultant at Cigital/Synopsys

- PhD from George Washington University
- Mother
- Ballroom dancer
- Twitter @KseniaDmitrieva

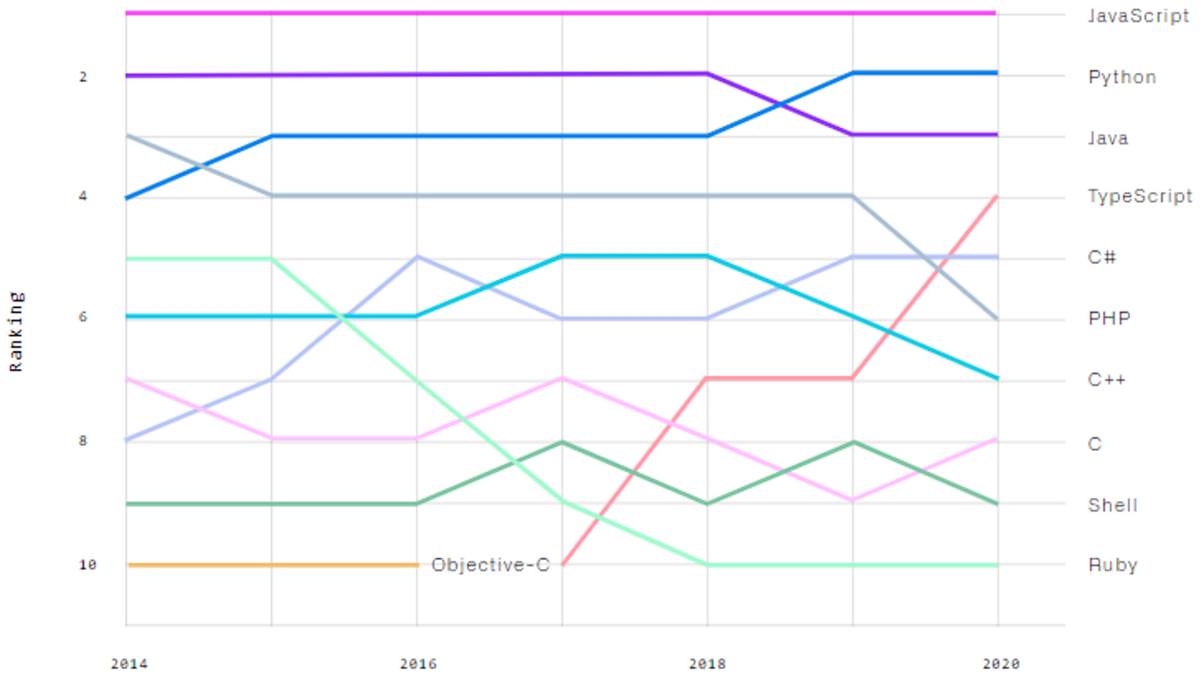
Agenda

1. Background on JavaScript frameworks
2. Client-side JavaScript frameworks and XSS
3. Server-side JavaScript frameworks and CSRF
4. Electron framework and desktop application vulnerabilities
5. Conclusion

Popularity of JavaScript

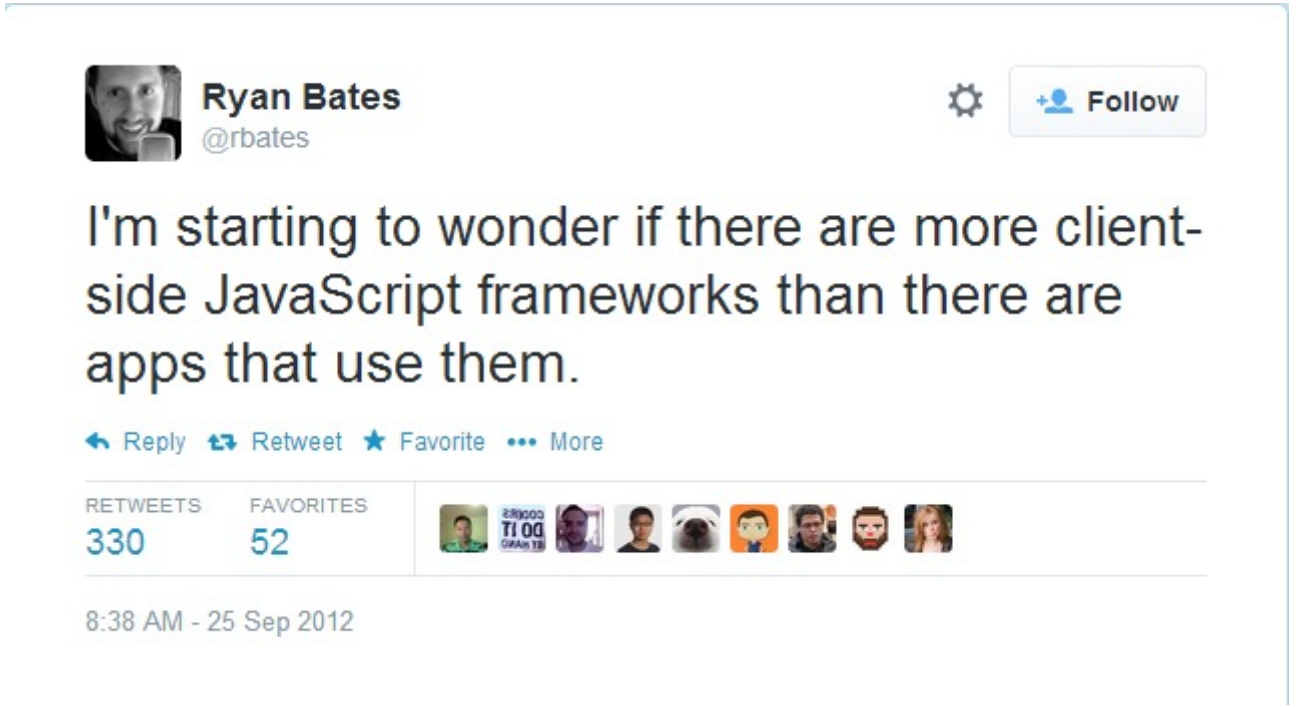
Language popularity by open pull request according the GitHub's Octoverse report from 2014 to 2020:

- JavaScript has been the leading programming language for the last 7 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



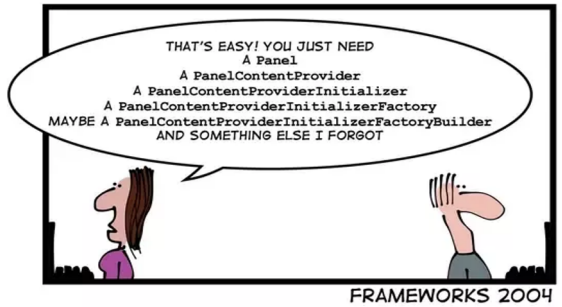
How many frameworks are there?

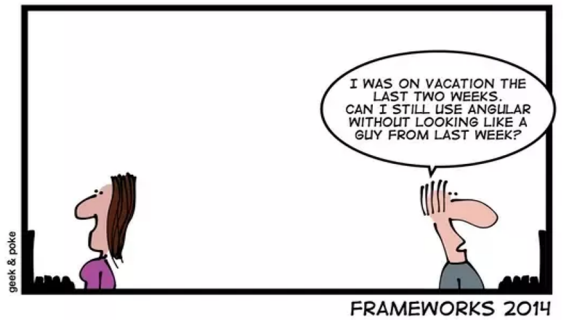
Frameworks and application development:

- According to Open Source Security and Risk Analysis (OSSRA) report in 2019, 70% of the analyzed applications was made up by open source code. Large part of that is frameworks.

How many frameworks are there in JavaScript ecosystem?

- 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

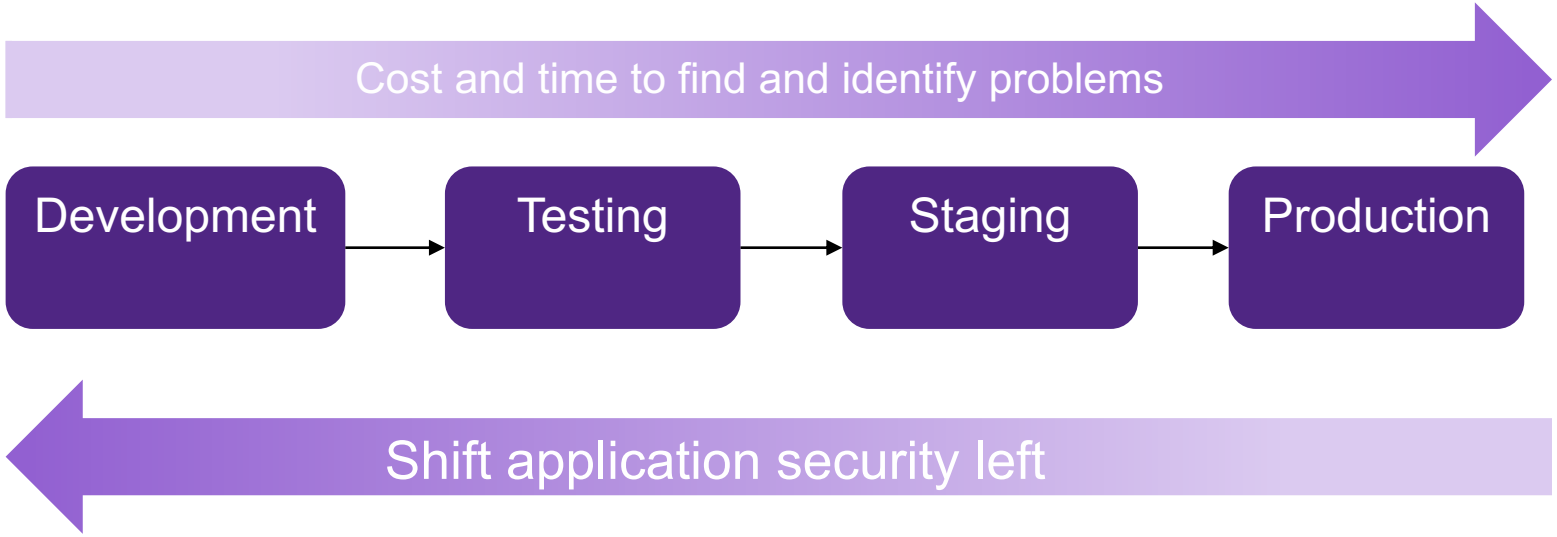




What is there in the framework for security?

- 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?

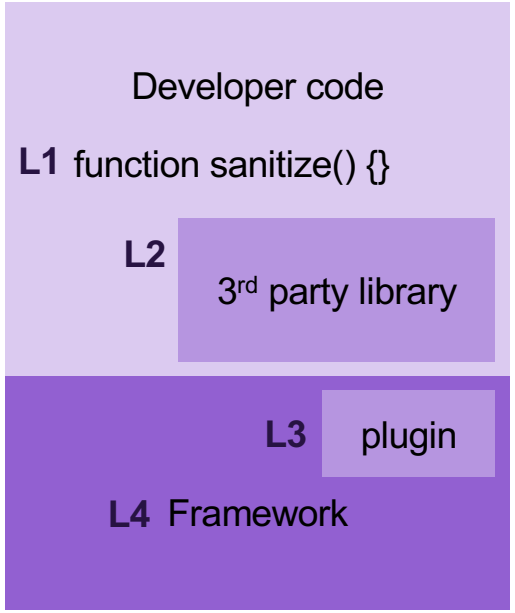
- Does building security controls into a framework result in “shifting-left” the security of the application?

Levels of Vulnerability Mitigation

proposed by John Steven

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

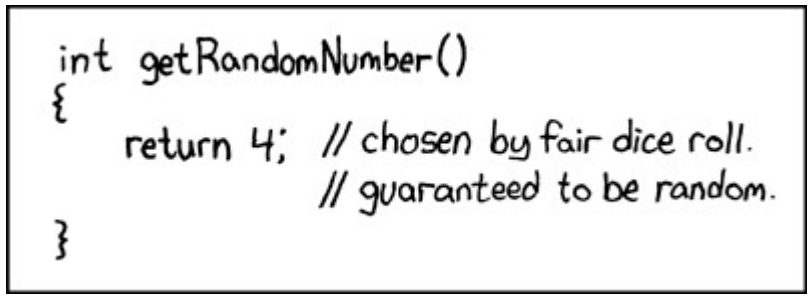
- **L0 - No mitigation in place.** Baseline – no protection
- **L1 - Custom function.** A sanitization routine written by developers
- **L2 - An external library** that provides a sanitization function
- **L3 - A framework plugin.** A third-party code used by developers which tightly integrates with 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.

Client-side JavaScript frameworks and XSS

Case Study 1

Data Selection for XSS Study (2016)

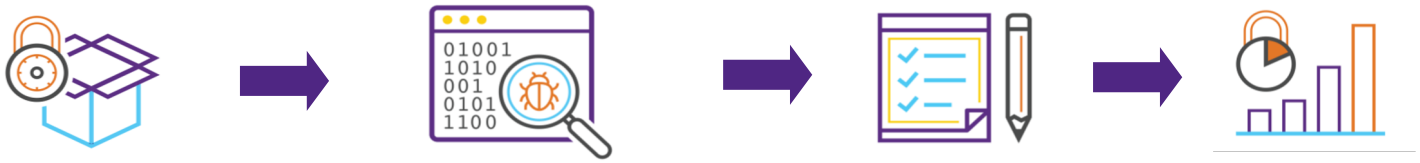
- Use case: the application needs to display user input that contains HTML markup

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

Total of 170 projects:

- 65 Jade/Pug
- 54 EJS
- 51 AngularJS

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

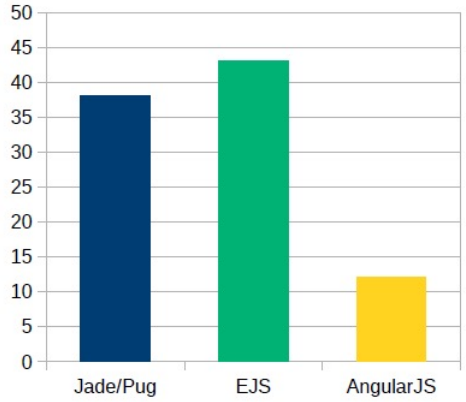
Perform manual review

Perform statistical analysis of the results

Case Study 1: 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

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

Server-side JavaScript frameworks and CSRF

Case Study 3

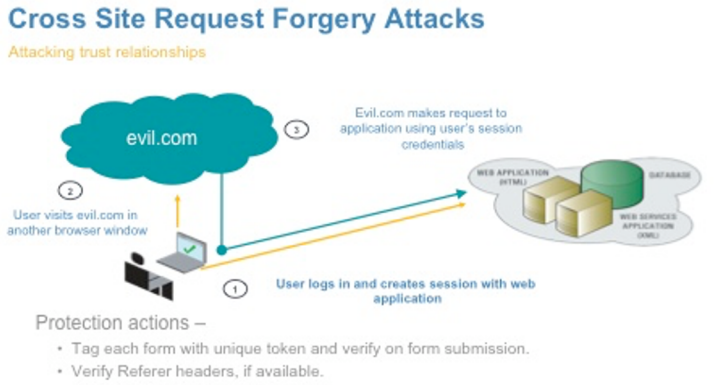
Case Study 2: CSRF

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



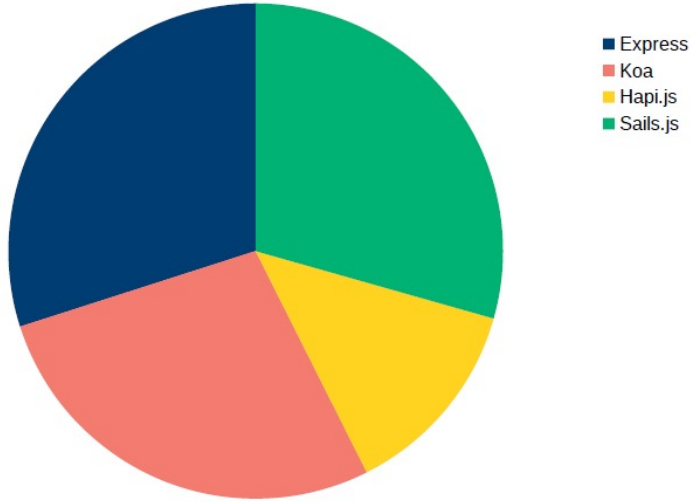
https://linuxsecurityblog.com/2016/02/11/defending-against-csrf-attacks/

Data Selection for CSRF Study (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
- Selected total 364 applications

Framework	Blog	CMS	E-commerce	REST API	Total	Mitigation Level
Express	29	35	45	0	109	L3
Koa	68	26	6	0	100	L3
Hapi	26	3	9	10	48	L3
Sails	72	20	15	0	107	L4

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):

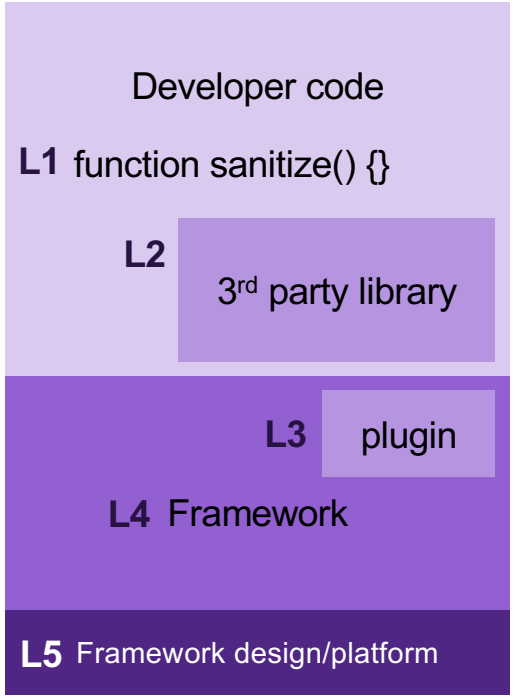
- Developed as access tokens, but used as session tokens
- Not stored in cookies, but transmitted in HTTP headers, which are not added to cross-origin requests by the browser
- Have other limitations, but do protect from CSRF

Levels of Vulnerability Mitigation

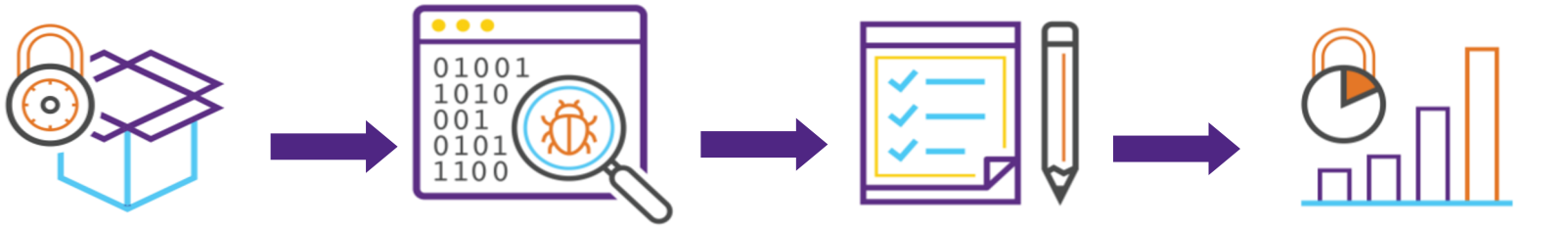
proposed by John Steven

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 sanitization routine written by developers
- **L2 - An external library** that provides a sanitization function
- **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



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

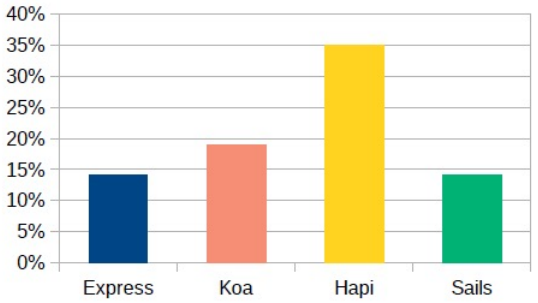
Perform manual review

Perform statistical analysis of the results

Case Study 2: Results

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

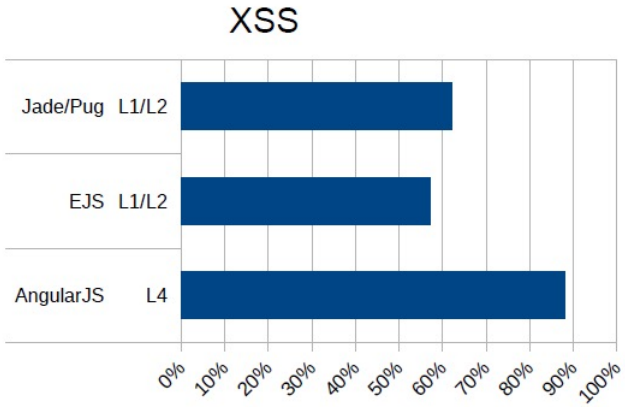
L3 - A framework plugin

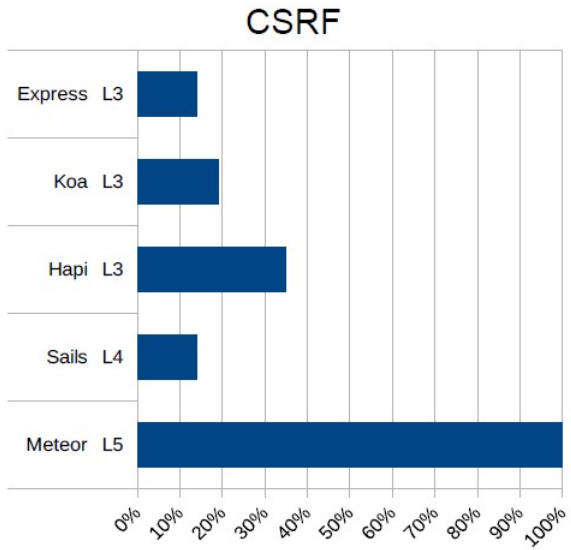
L4 - Built-in mitigation control

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.

Comparing XSS and CSRF Results

- Compare the percentage of protected projects by mitigation level/framework:





Why?

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

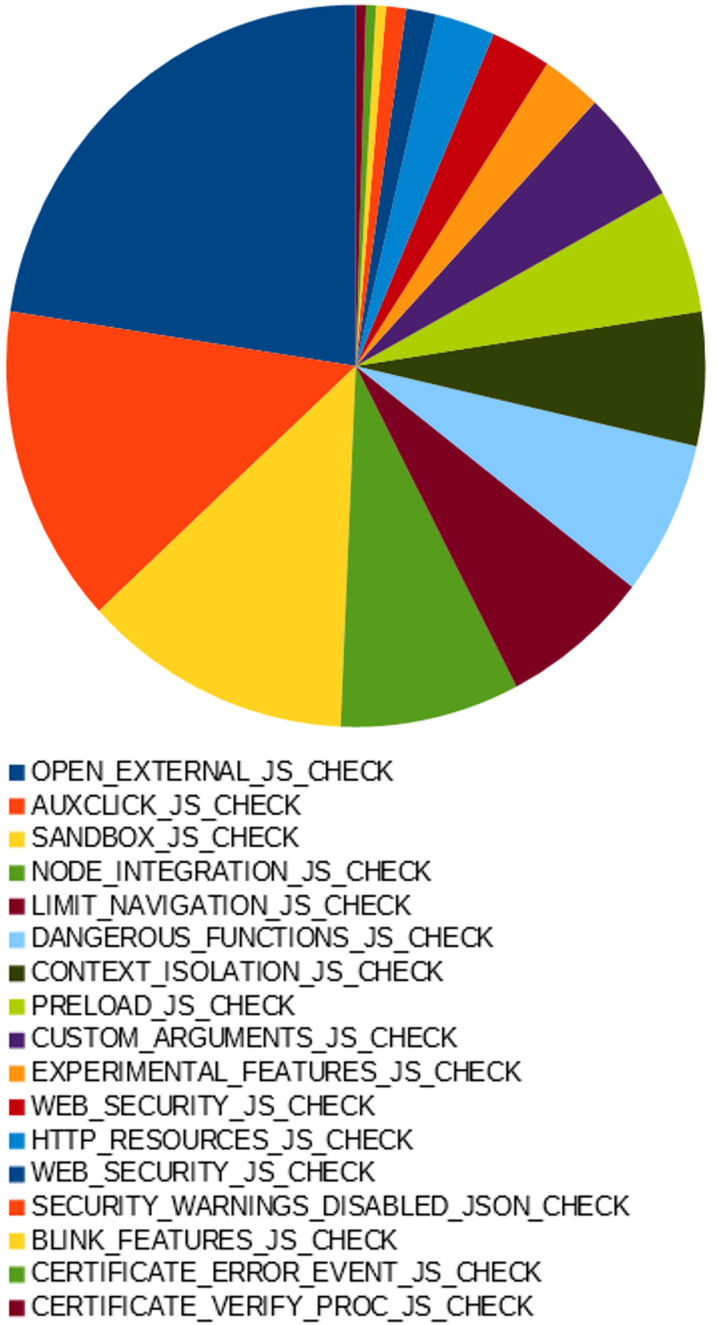
➢ Secure defaults are as important as the implementation levels of security controls

Electron and desktop application vulnerabilities

Case Study 3

Study of Electron Applications

- Selected 141 open source applications from Github, based on Awesome Electron https://github.com/sindresorhus/awesome-electron
  - Markdown editors
  - Messenger apps
  - Database clients
  - Password generators
  - Music players
- Ran Electronegativity and manually analyzed results https://github.com/doyensec/electronegativity
- Identified top 8 most common vulnerability categories out of 17 total categories



Electron Applications Study Results

- 141 open source applications
- 1680 total defects found automatically
- 464 findings were best practices, not leading directly to vulnerabilities > discarded
- 1216 potential vulnerabilities left
  - 218 true positives
  - 998 false positives
- Average defect density 0.11%
- Maximum defect density 2.66%
- Limitations of Electronegativity:
  - AST based analysis only. No dataflow, not sources or sinks. Leads to a lot of FPs and some FNs
  - No constant propagation (if a value is set to a variable, no defect will be discovered)

FP defect reported, because the value of sandobx1 is unknown

```
const { BrowserWindow } = require('electron')
let sandbox1 = true;
let win = new BrowserWindow({
  webPreferences: {
    nodeIntegration: false,
           sandbox: sandbox1
  }
})
win.loadURL(url)
```

Most Common Vulnerability Types

Vulnerability Type	Occurrence
OPEN_EXTERNAL_JS_CHECK	49
AUXCLICK_JS_CHECK	31
SANDBOX_JS_CHECK	27
NODE_INTEGRATION_JS_CHECK	15
LIMIT_NAVIGATION_JS_CHECK	15
DANGEROUS_FUNCTIONS_JS_CHECK	15
CONTEXT_ISOLATION_JS_CHECK	13
PRELOAD_JS_CHECK	12

Can some of them be mitigated by changing the Electron framework?

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

Built-in Security Controls in Electron

- **`nodeIntegration`**
  - Renderer process has access to Node.js APIs by default (e.g. `require()`, `fs` module, etc.)
  - Need to limit Node.js APIs from the content loaded externally
  - In v. 5.0.0 the `nodeIntegration` setting was changed to “false” by default
- **`sandbox`**
  - By default Chromium sandbox is disabled to allow Renderer code to access Node.js API, native Electron API, third-party modules
  - Additional protection if `nodeIntegration` is circumvented
  - Disabled by default
- **`contextIsolation`**
  - Allows to isolate JavaScript execution context between the main process and the renderer process
  - Attack: override built-in JavaScript methods through prototype pollution and then toggle the method call
  - In v. 5.0.0 proposed to enable it by default, but was implemented in v. 12.0.0

Most Common Vulnerability Types and Mitigations

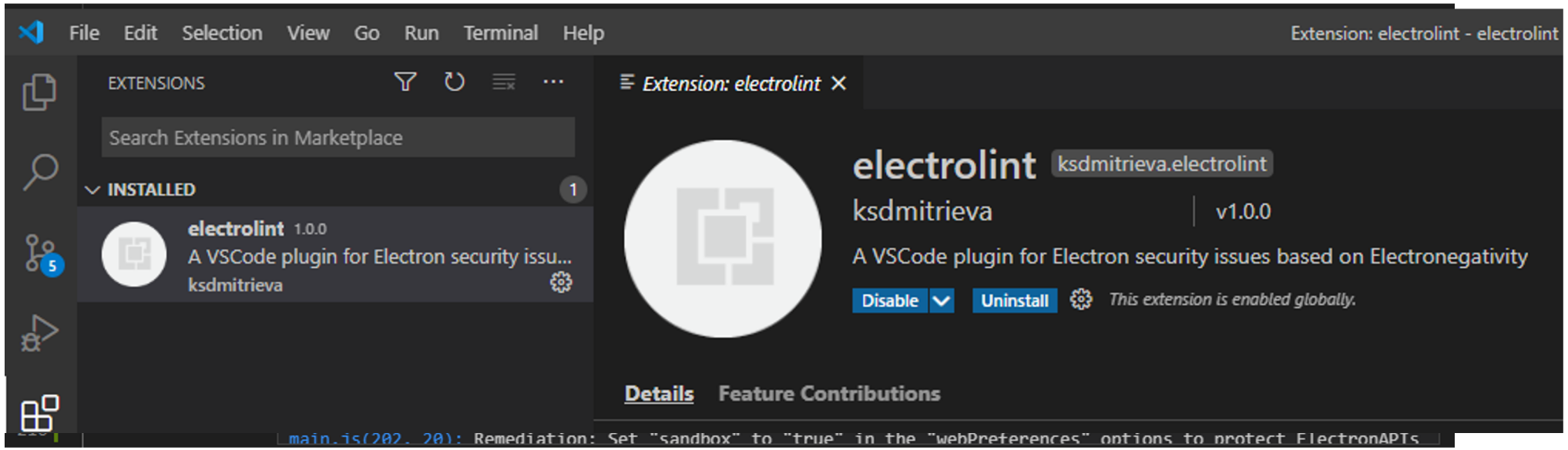
Vulnerability Type	Occurrence	Mitigation Level	Description
OPEN_EXTERNAL_JS_CHECK	49	4	Additional API
AUXCLICK_JS_CHECK	31	4	Secure default
SANDBOX_JS_CHECK	27	4	Secure default
NODE_INTEGRATION_JS_CHECK	15	4	**Secure default***
LIMIT_NAVIGATION_JS_CHECK	15	4	Policy control
DANGEROUS_FUNCTIONS_JS_CHECK	15	No, 1 or 0	No suggestion
CONTEXT_ISOLATION_JS_CHECK	13	4	**Secure default***
PRELOAD_JS_CHECK	12	No, 1 or 0	No suggestion

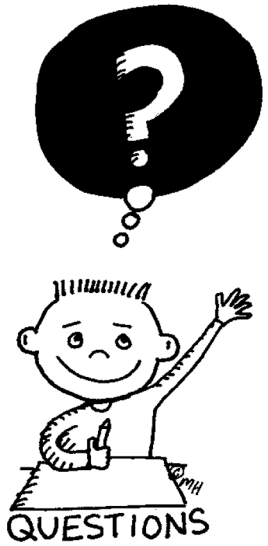
What should a developer do?

- It takes a long time to fix the framework – not an option
- Provide useful tools to developers early in the life cycle

Check it out: https://github.com/ksdmitrieva/electrolint
Use and contribute!

- We created a VisualStudio Code plugin **Electrolint**
  - Scans the code with Electronegativity
  - Highlights the vulnerable source code in the IDE
  - Provides **contextual mitigation** for the top 8 common vulnerabilities and more





Ksenia Peguero
ksenia@synopsys.com
Twitter: @KseniaDmitrieva

https://github.com/ksdmitrieva/electrolint