Baby steps towards the precipice

How the web became a scary place

... and how we can fix it

Artur Janc (aaj@google.com), USENIX Security 2019


Serious Sports Fans Only $1,000,000 in Cash and Prizes!
For serious sports fans only! Play Fantasy Football!
LYCOS
It's amazing where Go Get It will get you.
Find: rednex
Go Get It
Enhance your search.
NETSCAPE Now! 3.0
INVEST $100K
FREE Microsoft Internet Explorer
New Search . TopNews . Sites by Subject . Top 5% Sites . City Gu
PeopleFind . Point Review . Road Maps . Software . About Lyco
Add Your Site to Lycos
Copyright © 1996 Lycos™, Inc. All Rights Reserved.
Lycos is a trademark of Carnegie Mellon University.
Questions & Comments
Become a Rednex member NOW !!!
Check out loads of songs/videos - FREE!!
BUY REDNEX!!!
POP BAND FOR SALE
Now only $ 2.900.000 !!! ... and it's all yours!
REDNEX
Welcome to the official site for Rednex, famous for No.1 hits such as Cotton Eye Joe, Wish You Were Here, Spirit Of The Hawk and Old Pop In An Oak.
Login (e-mail)
Password
GO
NEW RELEASE !
REDNEX
- Devil's On The Loose -
Released January 7th.
Check it out HERE !!!
News
Pictures
Music
Sign up!
Videos
Download & Shop
Links
Biography
Shop
Donation
Book Rednex Here!
Lyrics & Credits
Discography
All Band Members & V I P's
Press & Radio
Tour Schedule
We rocked at I love the 90's in Hasselt, Belgium!
We recently performed in HASSELT, BELGIUM for 22,000 crazy party goers at the Arena party of the year!

Scaling application security at Google

The largest web application ecosystem in the world:

- 1,376 distinct user-facing applications on 602 *.google.com subdomains
- Thousands of internal apps, hundreds of acquired companies

… built using a wide variety of technologies:

- 4 major server-side languages: Java, C++, Python, Go
- 16+ HTML template system engines, dozens of HTML sanitizers
- JS & TypeScript with many frameworks: Angular, Polymer, Closure, GWT
- Over 2 billion lines of code, thousands of third-party libraries

… receiving thousands of web security vulnerability reports each year.

The web then:

“We should work toward a universal linked information system, in which generality and portability are more important than fancy graphics techniques and complex extra facilities. (...) The aim would be to allow a place to be found for any information or reference which one felt was important, and a way of finding it afterwards. The result should be sufficiently attractive to use that it the information contained would grow past a critical threshold, so that the usefulness the scheme would in turn encourage its increased use.”

- Tim Berners-Lee, March 1989

The web now:

Millions of applications, billions of users, trillions in market cap, zettabytes of data


PWAs
HTML5
Rise of JS frameworks
Web 2.0
First Browser War
Netscape & "E-Commerce"
HTML & HTTP/0.9

ap·pli·ca·tion plat·form

/ˌapləˈkāSH(ə)n ˈplatfôrm/

a framework of services that application programs rely on for standard operations

web plat·form

/web ˈplatfôrm/

the set of features implemented in a web browser used by developers to create web applications, e.g. HTML, CSS, JavaScript, network & storage APIs, etc

web bug

/web bəg/

a vulnerability which allows the disclosure or modification of data in a web application, exploitable against a logged-in user

The rest of this talk

1. **The insecurity of web application code**
   aka Why it's so hard for developers to write secure webapps

2. **The quagmire of legacy web features**
   aka The problems with the web's current security boundaries

3. **The dangerous land of new web APIs**
   aka A few notes on eternal vigilance

An incomplete list of people whose ideas are featured in this talk

@mikewest

@annevk @TanviHacks @fugueish @kneecaw @johnwilander @kkotowicz @we1x @sirdarckcat @empijei @mikispag @slekies @lcamtuf @_tsuro @lukOlejnik @emschec @dveditz @frgx @sleevi_ @jmhodges @pdjstone @EdFelten @jruderman

Charlie Reis, Łukasz Anforowicz, Nika Layzell, Christoph Kerschbaumer, Yutaka Hirano, Ryosuke Niwa, Christoph Kern

Note: They may not like this talk. Except Mike.

Part I

Insecurity of web application code

GOOGLE VULNERABILITY REWARD PROGRAM

2018 Year in Review

$3.4 MILLION
TOTAL REWARDS IN 2018

$1.7 MILLION
REWARDED FOR ANDROID AND CHROME VULNERABILITIES

MORE THAN
$15 MILLION
TOTAL REWARDS SINCE THE PROGRAM WAS FOUNDED IN 2010

1,319
INDIVIDUAL REWARDS

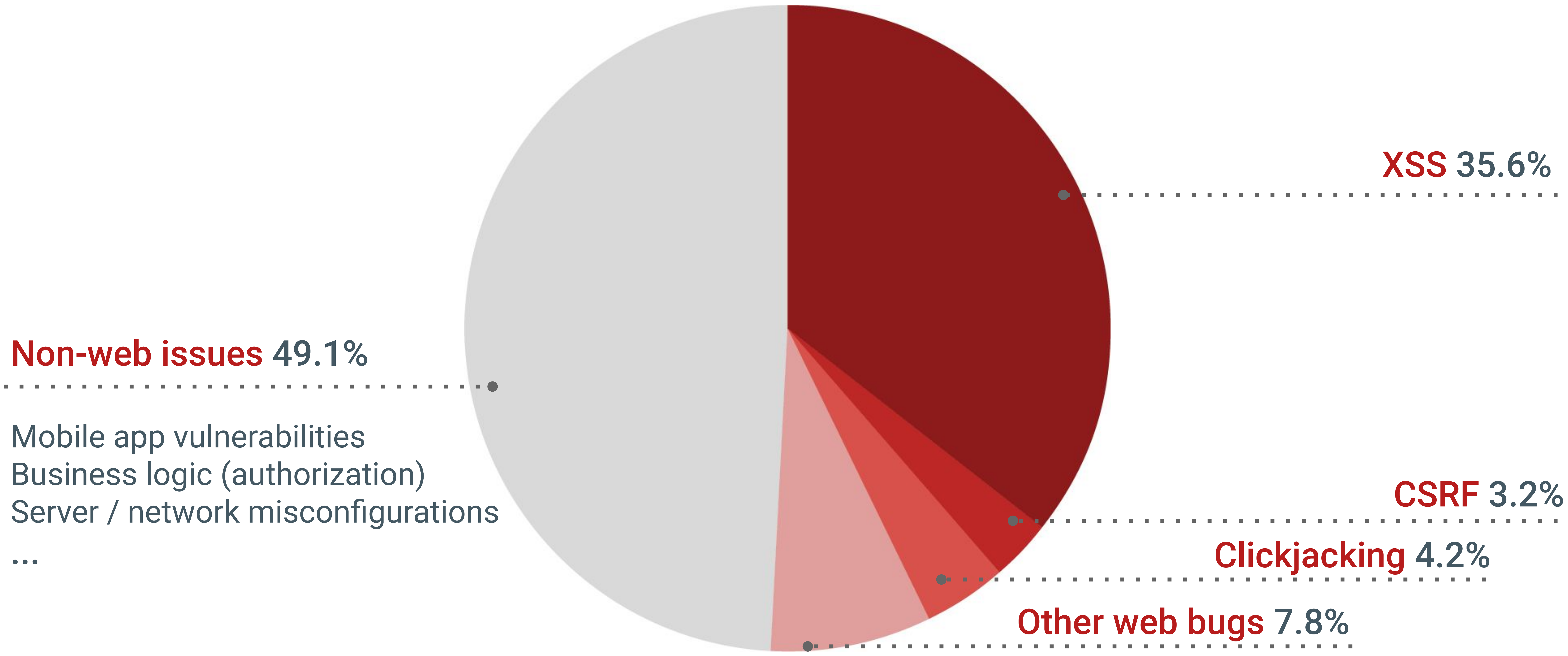
317
PAID RESEARCHERS

78
COUNTRIES REPRESENTED IN BUG REPORTS AND REWARDS

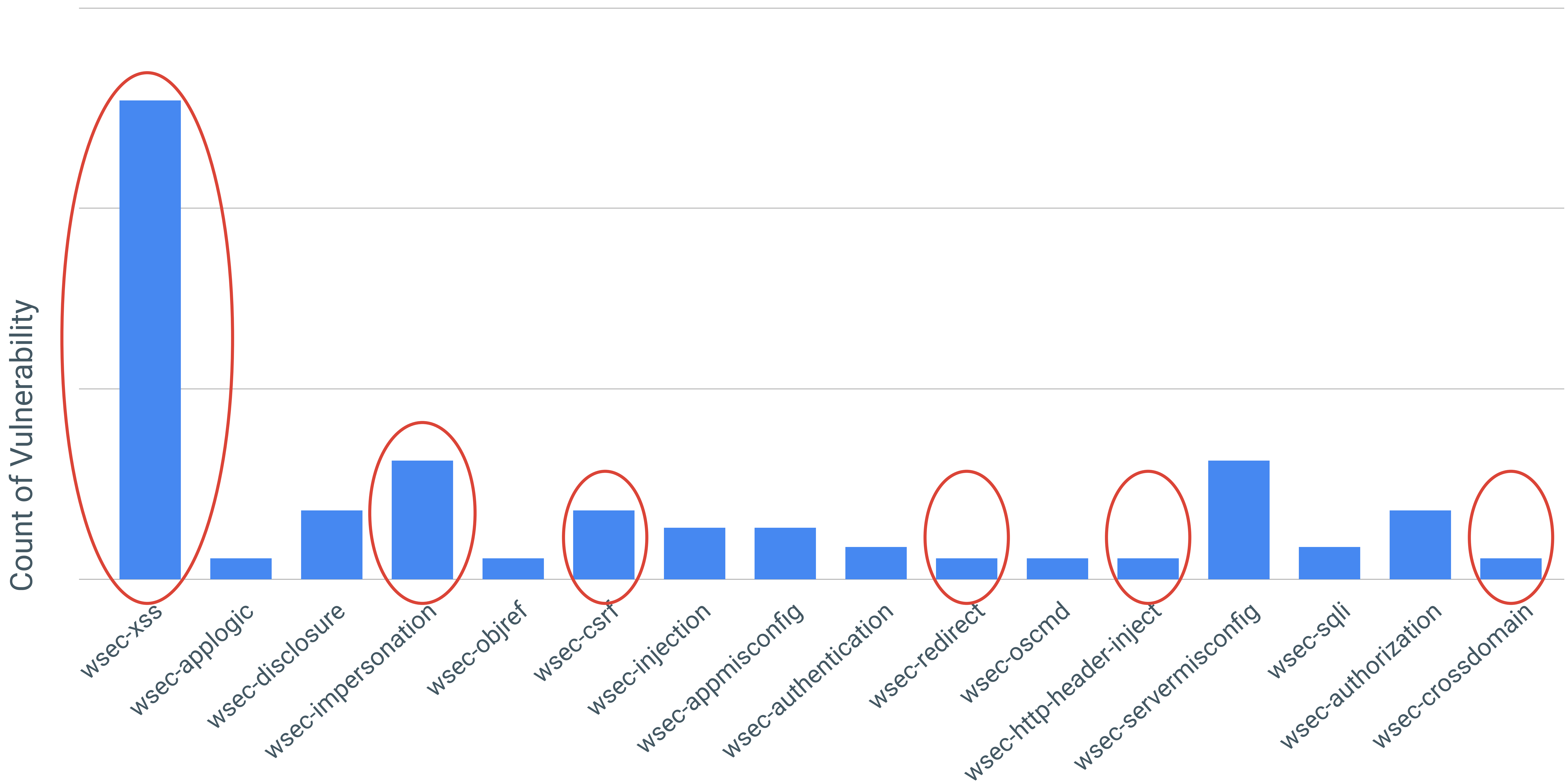
$41,000
BIGGEST SINGLE REWARD

$181,000
DONATED TO CHARITY

Total Google Vulnerability Reward Program payouts in 2018



Paid bounties by vulnerability type on Mozilla websites in 2016 and 2017



Source: @jvehent, Mozilla

HackerOne: Vulnerabilities by industry

	Consumer Goods	Financial services & insurance	Government	Healthcare	Media & Entertainment	Professional services	Retail & Ecommerce	Technology	Telecom	Transportation	Travel & Hospitality
Cross Site scripting (XSS)	23%	24%	26%	19%	28%	27%	24%	21%	24%	59%	38%
Information disclosure	17%	18%	18%	25%	16%	14%	16%	30%	18%	1%	13%
Improper authentication	7%	8%	3%	6%	9%	11%	8%	8%	5%	18%	10%
Violation of secure design principles	6%	9%	11%	10%	10%	12%	9%	8%	13%	6%	4%
Cross-site request forgery (CSRF)	12%	10%	4%	8%	7%	5%	12%	7%	8%	2%	8%
Open redirect	4%	6%	8%	5%	7%	6%	8%	5%	4%	2%	9%
Privilege Escalation	5%	4%	1%	1%	3%	5%	5%	5%	10%	3%	6%
Improper access control	12%	9%	3%	9%	6%	7%	8%	6%	5%	2%	4%
Cryptographic issues	2%	2%	18%	1%	2%	2%	1%	2%	3%	1%	1%
Denial of service	2%	2%	1%	1%	1%	2%	1%	2%	2%	1%	1%
Business logic errors	4%	5%	1%	4%	5%	6%	4%	4%	3%	2%	5%
Code injection	1%	1%	1%	5%	2%	2%	2%	2%	2%	1%	1%
SQL injection	5%	1%	5%	4%	2%	0%	2%	2%	2%	2%	1%
	1%	1%	1%	2%	1%	1%	1%	1%	2%	1%	1%
	1%	1%	0%	0%	1%	0%	1%	1%	1%	1%	0%

Source: HackerOne report, 2018

Figure 5: Listed are the top 15 vulnerability types platform wide, and the percentage of vulnerabilities received per industry

HackerOne: Vulnerabilities by industry

	Consumer Goods	Financial services & insurance	Government	Healthcare	Media & Entertainment
Cross Site scripting (XSS)	23%	24%	26%	19%	28%
Information disclosure	17%	18%	18%	25%	16%
Improper authentication	7%	8%	3%	6%	9%
Violation of secure design principles	6%	9%	11%	10%	10%
Cross-site request forgery (CSRF)	12%	10%	4%	8%	7%
Open redirect	4%	6%	8%	5%	7%

Source: HackerOne report, 2018

The three cardinal sins of the web as an application platform

Sin #1

Mixing code and data

A few *completely safe* code examples

```
response.write("<h1>Hello, " + name);
```

```
element.innerHTML = name;
```

```
$('body').append(name)
```

```
<a href="{user.homepage}">homepage</a>
```

```
window.location = user.homepage;
```

```
response.write("Content-Type: text/csv")
response.write(user.dataExport)
```

A few ~~completely safe~~ code examples

```
response.write("<h1>Hello, " + name
```
XSS
```
<script>alert(1)</script>
```

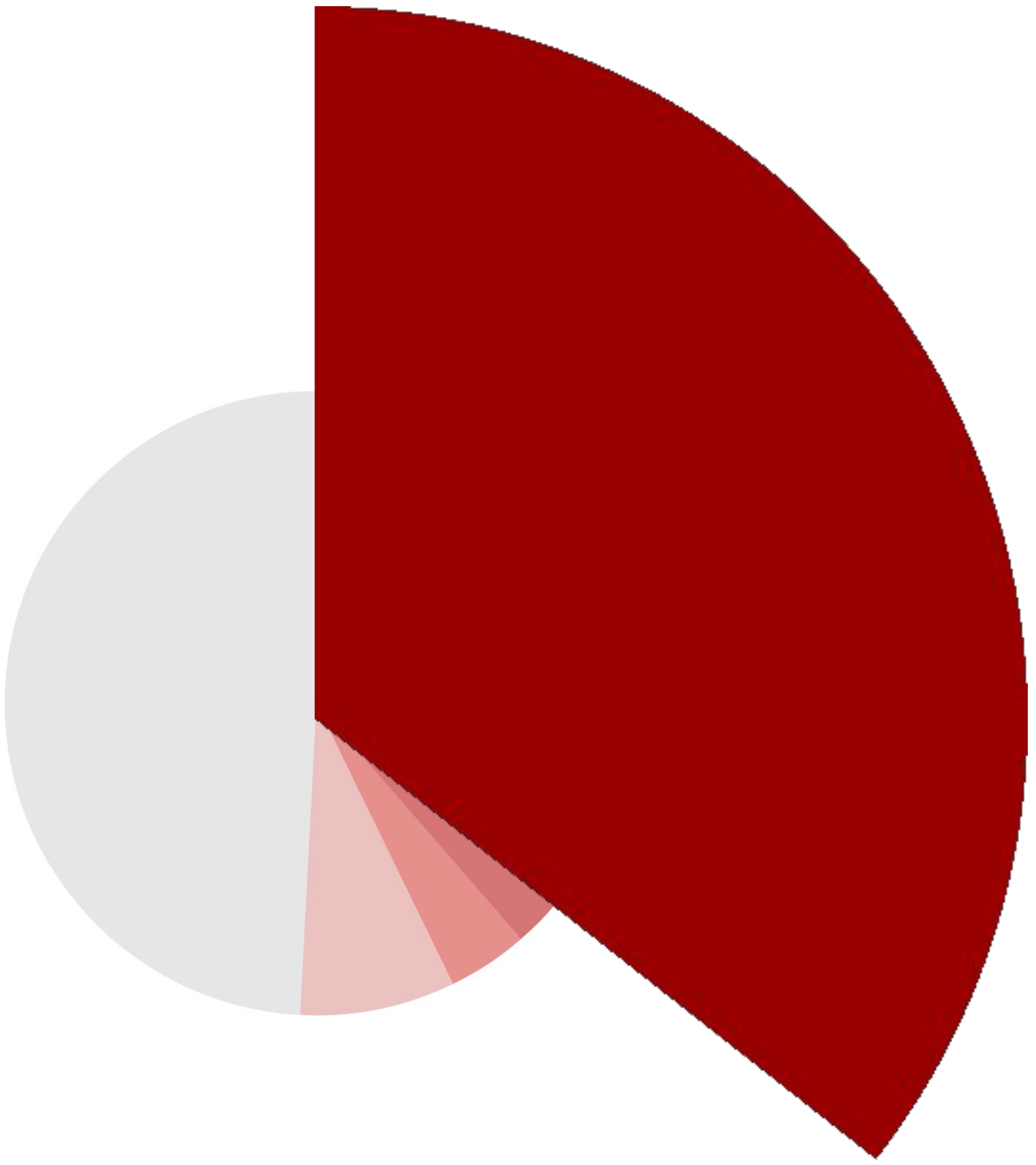
```
element.innerHTML = name;
```
XSS
```
<img src=. onerror=alert(1)>
```

```
$('body').append(name)
```
XSS
```
<img src=. onerror=alert(1)>
```

```
<a href="{user.homepage}">homepage<
```
XSS
```
javascript:alert(1)
```

```
window.location = user.homepage;
```
XSS
```
javascript:alert(1)
```

```
response.write("Content-Type: text/csv")
response.write(user.dataExport)
```
XSS
```
...<script>alert(1)</script>
```



Mixing code and data

Bugs: Cross-site scripting (XSS)

Two major problems:

1. It's easy to introduce XSS during the most mundane web development tasks: generating a part of the UI, creating links, serving files with any user-controlled data.
2. XSS is web-level **remote code execution**, giving the attacker full access to user data in the application.

Sin #2

Unrestricted attack surface

A few *completely safe* code examples

```
<form action="/transfer">
  <input name="target" value="frgx" />
  <input name="amount" value="10" />
```

```
<button onclick="deleteAccount()">
  Delete account</button>
```

```
w("Content-Type: text/javascript")
w("var data = {'user':'${name}'}")
```

```
if search_result:
  log_to_db(search_query);
  return search_result
```

A few ~~completely safe~~ code examples

```
<form action="/transfer">
  <input name="target" value="f
  <input name="amount" value="10" />
```

CSRF

```
<form action="//victim/transfer">
<input name="target" value="evil" />
<input name="amount" value="1000" />
```

```
<button onclick="del
  Delete account</bu
```

clickjacking

```
<iframe src="//victim/settings"
    style="opacity: 0"></iframe>
```

```
w("Content-Type: text/javascri
w("var data = {'user':'${name
```

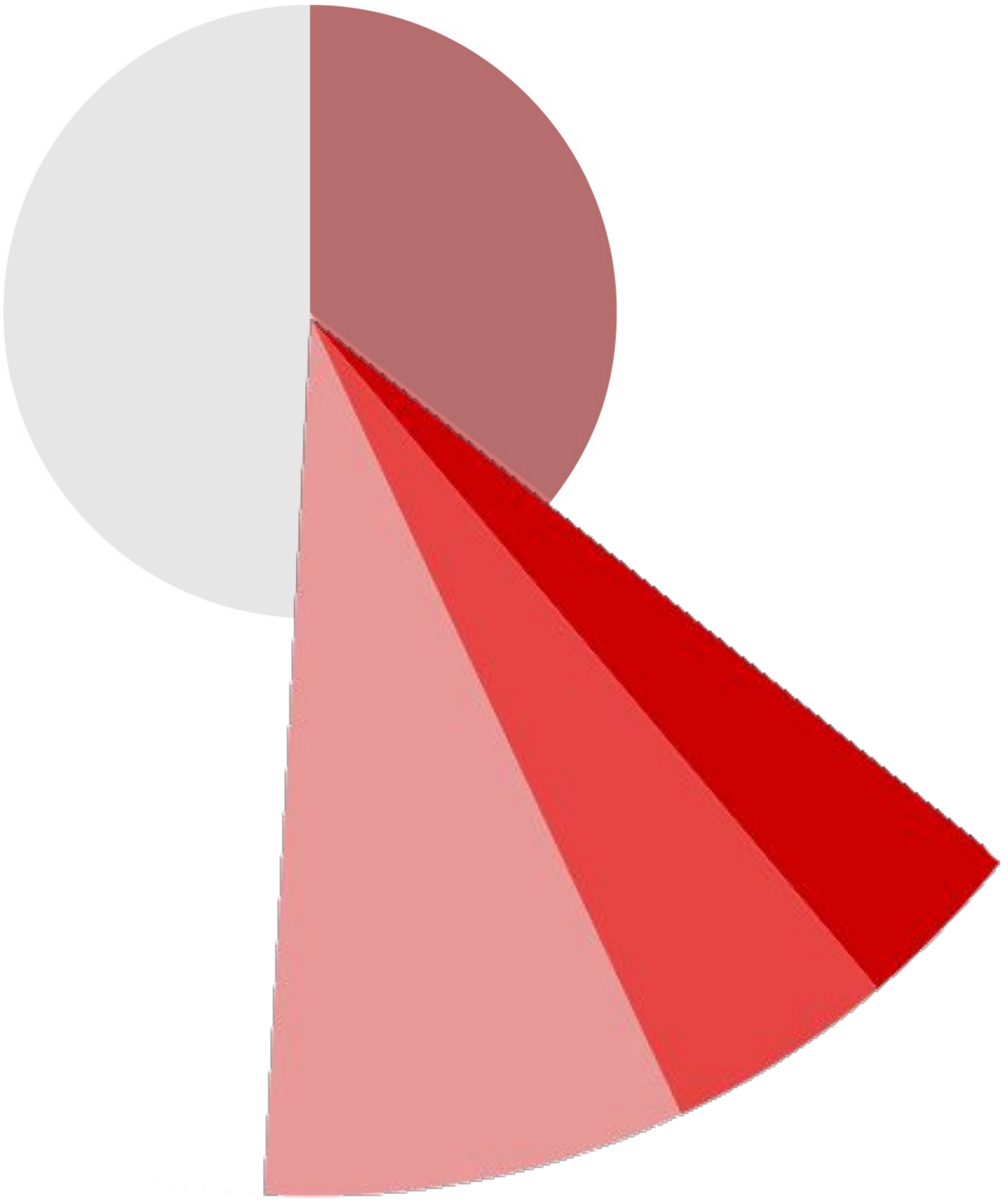
XSSI

```
<script src="//victim/json" />
<script>alert(data)</script>
```

```
if search_result:
  log_to_db(s
  return search_result
```

XS-Search / timing

```
<script>t=performance.now()</script>
<img src="//victim/search?q=secret"
  onerror="t2=performance.now()" />
```



Unrestricted attack surface

Bugs: Cross-site request forgery (CSRF), cross-site script inclusion (XSSI), clickjacking, cross-site search (XS-Search), timing attacks

The problem:

1. Every application endpoint is addressable via a URL.
2. Every request automatically attaches application cookies, regardless of who sent the request.

The result: Lack of real isolation between applications.

Sin #3

Insecure transport layer

<plain>

A few *completely safe* configuration examples

```
<VirtualHost *:80>
    DocumentRoot "/www/example"
    ServerName victim.example
</VirtualHost>
```

```
response.write("Set-Cookie: ID=" + id)
```

<plain>

A few ~~completely safe~~ configuration examples

```
<VirtualHost *:80>
    DocumentRoot "/www/example"
    ServerName victim.example
</VirtualHost>
```

No encryption

```
response.write("Set-Cookie: ID=" + id)
```

Cookie sent over HTTP

<plain>

Insecure transport layer

Bugs: MitM (sslstrip), mixed content / scripting

The problem: For much of the web's existence hosting over unencrypted HTTP has been the default configuration.

Note: We're ignoring problems with the CA ecosystem here.

Some other *minor sins*

- ➔ Unsafe defaults
- ➔ APIs that allow developers to shoot themselves in the foot
- ➔ Lack of defense-in-depth
- ➔ No application-wide security configuration

How have we managed to get away with this?

A glorious history of tacitly ignoring problems

- ~~Blaming the developer~~ Developer education
    - OWASP Top 10
- Finding vulnerabilities before they get exploited by attackers
    - Code reviews, pentests, bug bounties
- Automated vulnerability scanning
    - Crawlers, web scanners, HTTP header checking tools
- Hardened layers of abstraction
    - Safe-by-default higher-level libraries
    - Compile-time restrictions, integration with developer tools

The lack of a central web authority facilitates and absolves inaction.

A radical idea: What if we fixed this?

<plain>

Fixes: Insecure transport layer

Learn from the Moar TLS project, [Emily Schechter @ Enigma 2017](#).

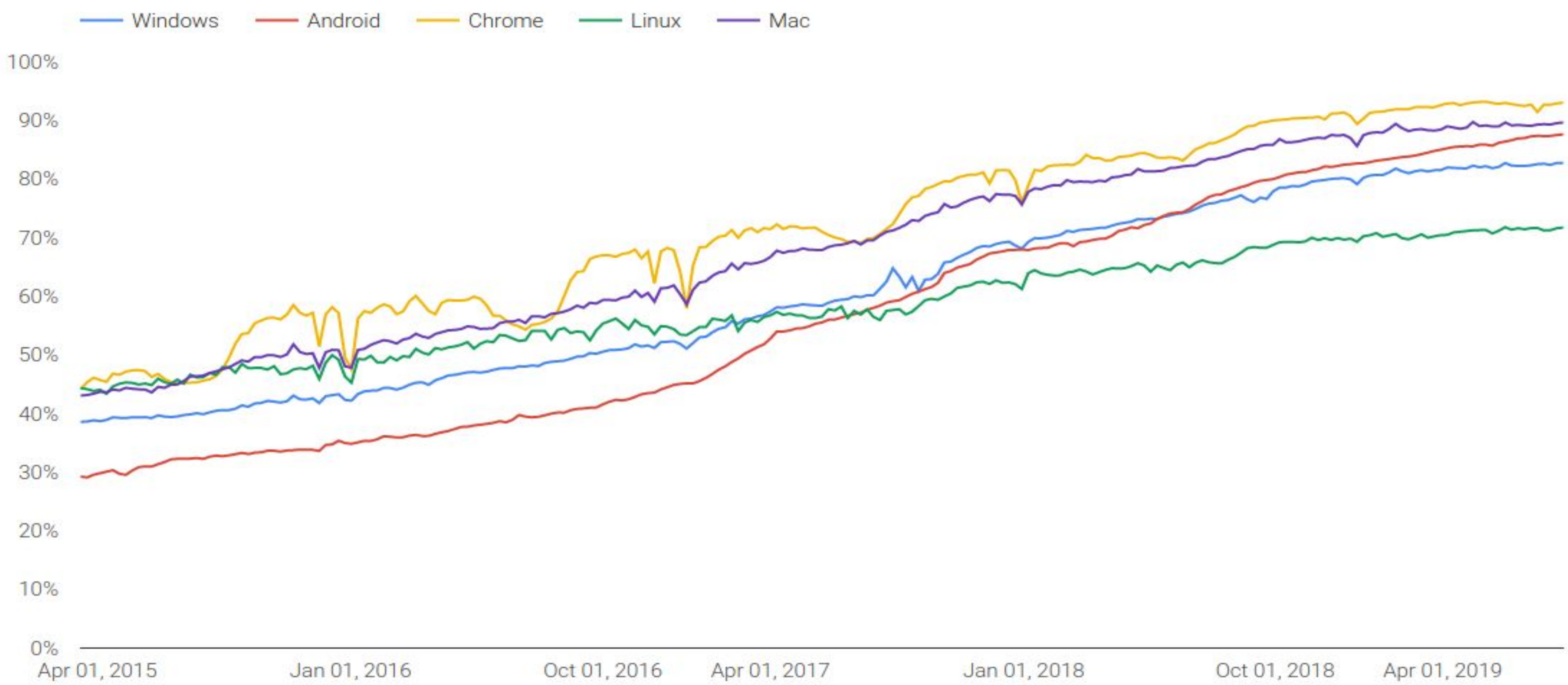
Prerequisite: The right platform security features must be available:

- HTTPS; the `Secure` attribute on cookies
- `HTTP Strict Transport Security` (HSTS) set via a header and preload list
- `'upgrade-insecure-requests'` in CSP3
- The `__Secure-` cookie prefix, cookie eviction rules
- Browser blocking of mixed content

Ecosystem support:

- LetsEncrypt as a free Certificate Authority
- [HTTPS encryption on the web](#) Transparency Report, outreach
- Browser UI work: ["Not secure"](#) for HTTP pages
- Certificate Transparency

% of pages loaded over HTTPS in Chrome by platform



Fixes: Mixing code and data

Prerequisite: Implement powerful features to protect from injections.

Server-side injections: Require a cryptographic CSP3 nonce/hash for every <script>

```
Content-Security-Policy: script-src 'nonce-random123'

<script nonce="random123">alert('this is fine!')</script>
<script>alert('This script is missing a nonce')</script>
```

Client-side injections: Make the DOM API safe by default with Trusted Types

```
Content-Security-Policy: trusted-types myPolicy

const SanitizingPolicy = trustedTypes.createPolicy('myPolicy', {
   createHTML(s: string) => myCustomSanitizer(s) });
el.innerHTML = SanitizingPolicy.createHTML(foo); // Needs TrustedHTML
```

Complement this with Origin Policy for origin-wide enforcement.

Fixes: Unrestricted attack surface

Prerequisite: Implement powerful general isolation features.

Allow applications to **reject untrusted cross-site requests**

- Annotate requests with source information using Fetch Metadata Request Headers

  `Sec-Fetch-Site` `Sec-Fetch-Mode` `Sec-Fetch-User`

- Adopt `SameSite` cookies and the Cross-Origin-Resource-Policy

Allow windows to **break references from cross-origin websites**

```
Cross-Origin-Opener-Policy: same-origin
```

```python
# Reject cross-origin requests to protect from CSRF, XSSI & other bugs
def allow_request(req):
  # Allow requests from browsers which don't send Fetch Metadata
  if not req['sec-fetch-site']:
    return True

  # Allow same-site and browser-initiated requests
  if req['sec-fetch-site'] in ('same-origin', 'same-site', 'none'):
    return True

  # Allow simple top-level navigations from anywhere
  if req['sec-fetch-mode'] == 'navigate' and req.method == 'GET':
    return True

  return False
```

Allowing developers to write safe applications

1. **Provide security mechanisms** to address injections and add isolation

   ... and ship them in all modern browsers, tomorrow.

2. **Help developers adopt them**
   - Write documentation and ship supporting features (Origin Policy)
   - Integrate with browser developer tools and HTTP header checkers

3. Start thinking about the path to **enable them by default**

   ... or **re-evaluate whether web-wide adoption should be the goal**

Note: Browser vendors need to do work, which may make them unhappy.

If we do this, will the web be a safe application platform?

Part II

The quagmire of legacy web features

Problem #1

The browser-enforced boundaries between web applications are fuzzy and imperfect.

A classic example: History detection

```
a:visited { color: red; }
a:link { color: blue}
```

→ visited
unvisited

```
> window.getComputedStyle(document.links[1]).color
<· "rgb(0, 0, 238)"
```

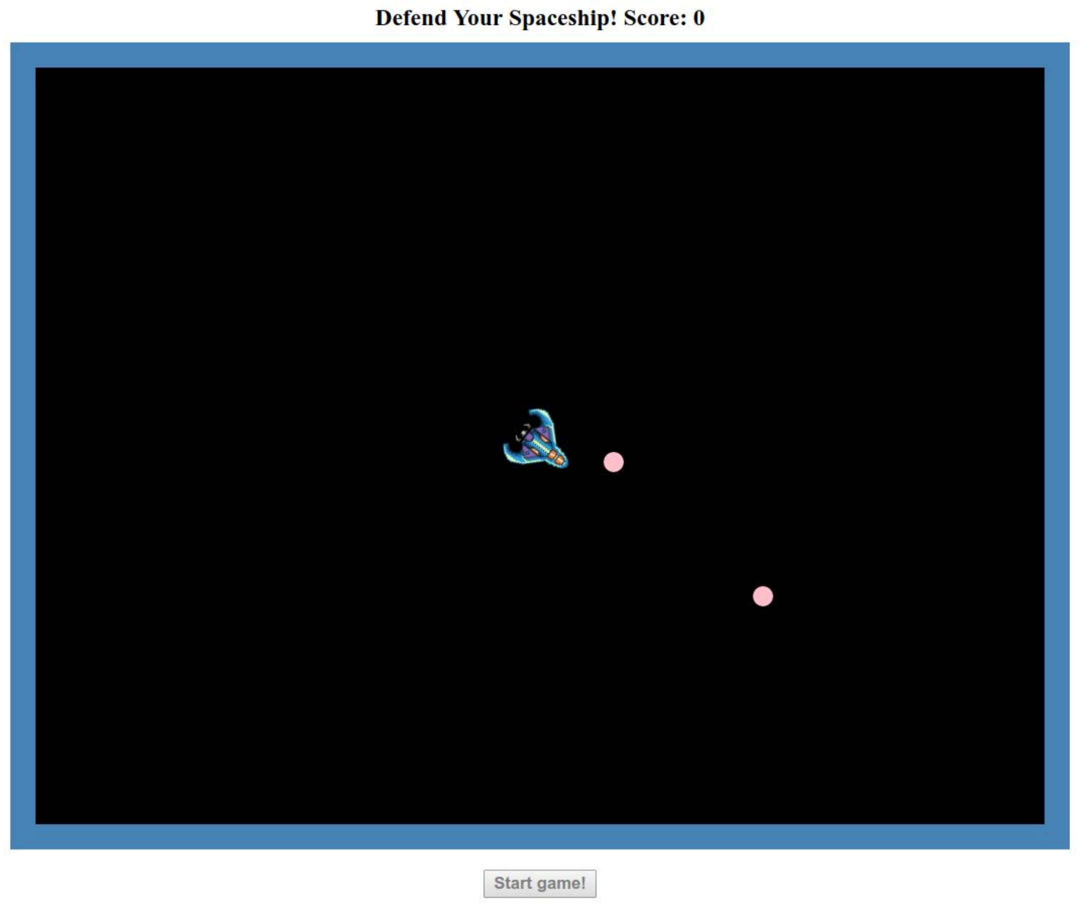
The problem: Sites can learn about all URLs visited by users, at high speed.

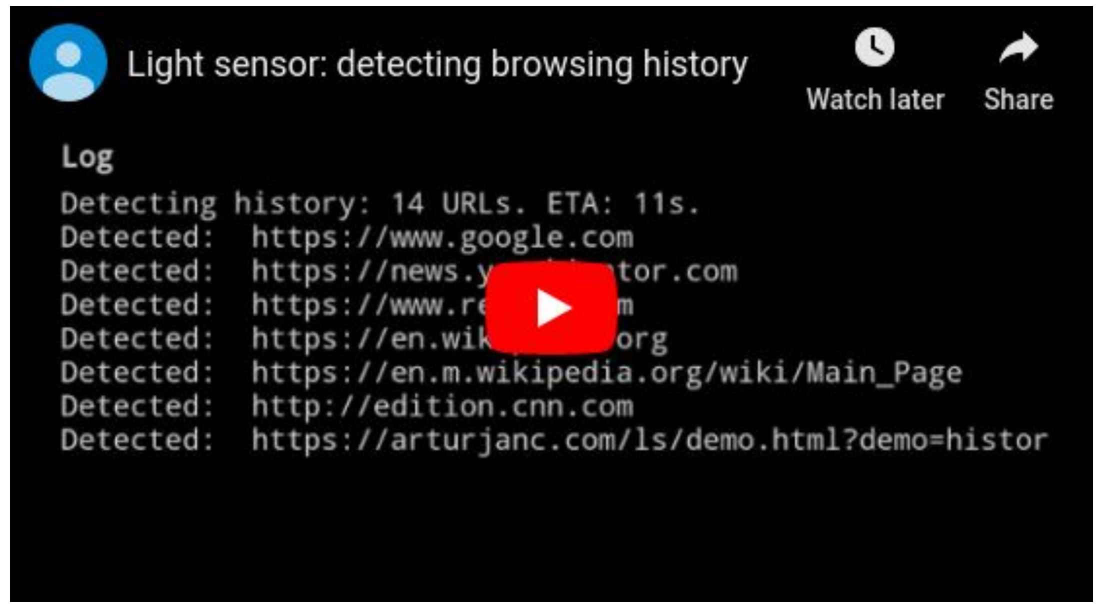
The fix (David Baron @ Mozilla):

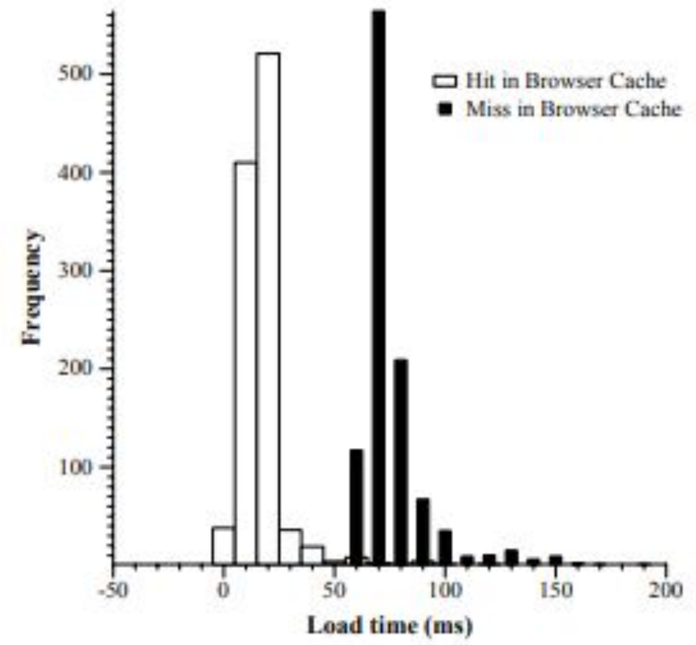
Make `getComputedStyle()` lie about the style of the link, limit CSS properties in `:visited` styles to those which don't affect layout, make CSS selectors treat all links as unvisited.

A classic example: History detection

- **Timing attacks** to detect color changes of a visited link:
  - #252165: Visited links can be detected via redraw timing
  - #835590: Complicated CSS effects and :visited selector leak browser history through paint timing
- **User interaction attacks**
  - Weinberg et al: I still know what you visited last summer
  - Michal Zalewski's "Asteroids" game
- Other quirky side channels
  - Screen color detected via the ambient light sensor







A classic example: Cache detection

Edward Felten, Michael A. Schneider: [Timing Attacks on Web Privacy]():

Use timing to detect resources in the cache, leaking browsing history.

Since then:

- More accurate **cache probing techniques**
  - More accurate timing APIs: `window.performance.now()`
  - Tricks: `window.stop()`, abortable fetch, clearing a URL from the cache
- More damaging **attacks**
  - XS-Search based on the presence of a cacheable resource:
    - https://victim.example/search?q=**foo** has `<img src="found.png">`
    - https://victim.example/search?q=**bar** doesn't have the image

Example: Connection exhaustion attacks

Stephen Roettger:
[Leak cross-window request timing by exhausting connection pool]

Attack:

- Establish `g_max_sockets_per_pool` connections to attacker's site
- Close one connection and navigate to https://victim.example
  - All fetches from the victim window happen via a single connection
- Make repeated requests to attacker's site.
  - Timing of requests reveals the timings of fetches in victim's window

Result: Timing attacks without making direct requests to the victim.

Example: Requests to local networks

Any website can make requests to services running on localhost and local networks: `192.168.0.0/16`, `172.16.0.0/12`, `10.0.0.0/8`

```
<img src="http://192.168.0.1/cgi-bin/…"></img>
<form method="POST" action="http://10.1.1.1/..."></form>
```

... and often get full script execution by using DNS rebinding.

Attacks:

- Reconnaissance on user's local network by using the browser as a proxy
- Exploiting vulnerabilities
  - CSRF bugs on routers, printers and IoT devices
  - Application servers exposed on localhost not expecting network traffic

Problem #2

We've accumulated many counter-intuitive features and APIs which cause problems.

Deprecate & remove web anti-patterns

A laundry list of terribleness accumulated over the years:

document.domain, MIME type sniffing, Referrer leakage, DOM clobbering, Public Suffix List, javascript: URIs, insecure transports (ftp://), file:// URI handling, (?) fingerprinting, ... , ... , ...

Similarly to Moar TLS, follow examples of successful Chrome efforts:

- Flash deprecation
- Deprecating powerful features on insecure origins

Fixing our past transgressions

1. **Clamp down on global state** to prevent information leaks
 - Remove :visited (or make it same-origin)
 - Add browser cache double-keying

   … and prevent network-level attacks with CORS and RFC1918

2. **Remove unsafe legacy APIs** and browser behaviors

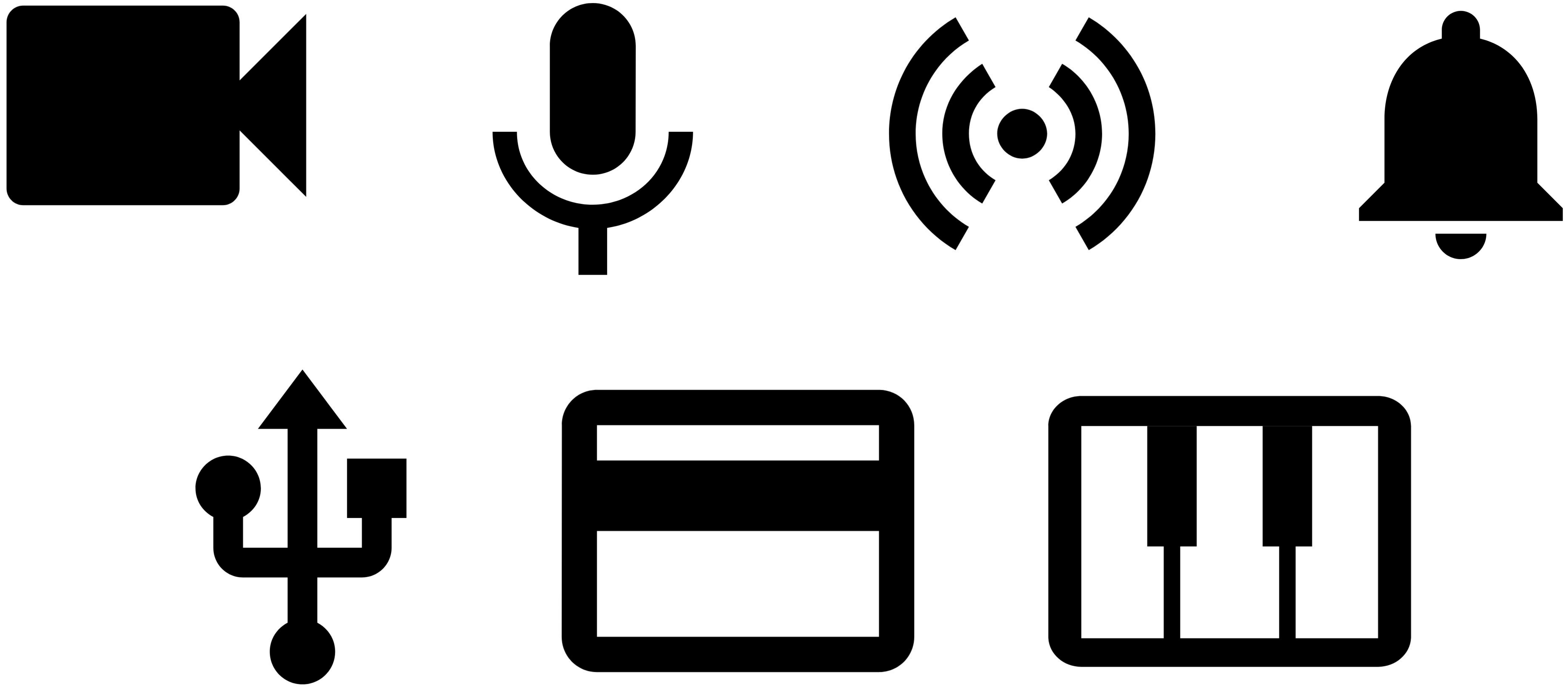
Note: This requires change. Change makes people unhappy.

Part III

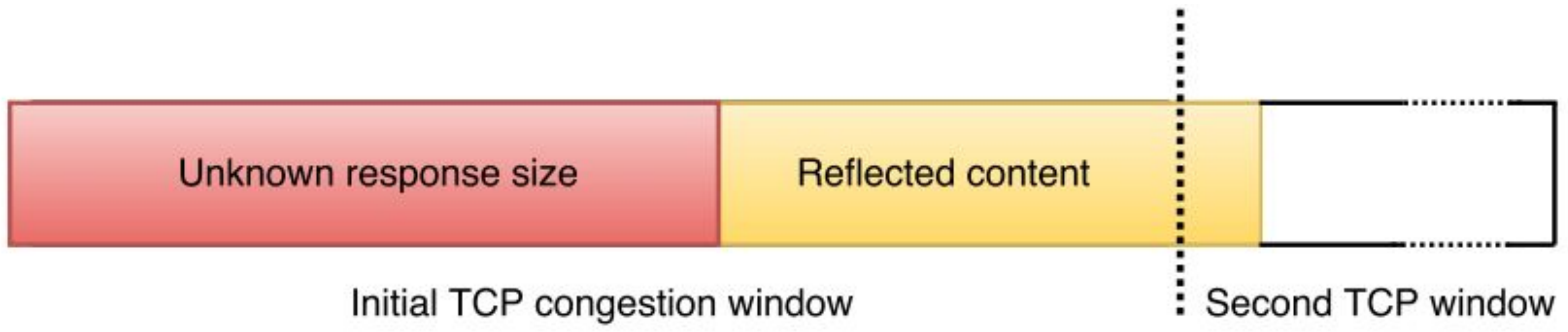
The dangerous land of new web APIs

The Problem

Any new API in the web platform changes the risk profile of *all* existing applications.



Example: HEIST

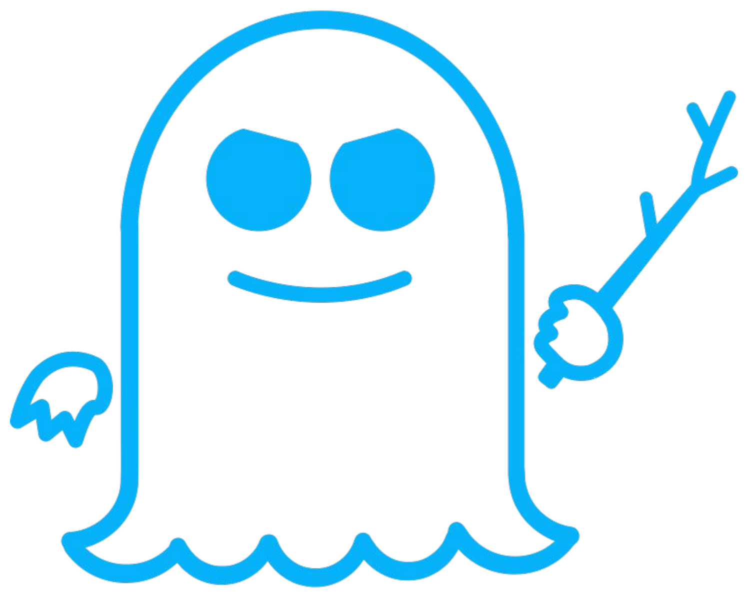


Mathy Vanhoef and Tom Van Goethem: [HEIST: HTTP Encrypted Information can be Stolen through TCP-windows]

Feature Δ: Fetch API resolves a Promise when 1st byte of response arrives.
Security Δ: The attacker can reliably determine if a response fits in a single TCP window by comparing Promise resolution to resource onload time.

Consequence: For responses with any user-controlled parts attacker can determine the size, allowing fully remote BREACH attacks to extract arbitrary secrets from the response.

Example: Spectre / Transient execution attacks



Paul Kocher, Jann Horn et al: Spectre Attacks: Exploiting Speculative Execution
Mark Seaborn: Multi-threading helps cache-based side channel attacks
Michael Schwarz et al: Fantastic Timers and Where to Find Them

Feature Δ: SharedArrayBuffer: arrays shared across threads, with atomicity.
Security Δ: Using a counting thread in a Web Worker allows the creation of a nanosecond-level timer, more accurate than explicit web timing APIs.

Consequence: Nanosecond-level timers allow practical exploitation of branch mispredictions on many CPUs, enabling the leaking of data from the process address space, revealing ~arbitrary cross-origin data.

Example: "scroll-to-selector" proposal

Current behavior: `https://victim/#foo` scrolls to `<div id="foo">`

Feature Δ: Allow the URL fragment to also scroll to an arbitrary CSS selector:

```
https://victim/#body>a[href^="https://"]
```

Security Δ: Attacker can force a scroll based on arbitrary HTML attributes.

Consequence: An attacker who controls an iframe on the target page can leak secrets from the DOM (e.g. CSRF tokens) by using sibling selectors and `IntersectionObserver` to detect if the iframe scrolled into view:

```
https://victim/#input[value="secret"] ~ iframe
```

Mitigating our inevitable future mistakes

1. **Review** new web platform APIs
   … and help browser developers design them safely

2. **Create new restrictive modes** which protect non-cooperating apps from the misuse of powerful / low-level APIs by potentially malicious sites:
- Require Cross-Origin-Embedder-Policy and Cross-Origin-Opener-Policy to unlock new APIs, e.g. threaded access to `SharedArrayBuffer`.

Note: This slows down features. Slowing down features makes people unhappy.


Fixing the web

It's simple. We need to:

… **implement** powerful new security features

… **remove** bad old legacy features

… **review** and contain new platform features

Thank you! / Q&A