

# How to Reverse Engineer Web Applications



# First things first

## Prerequisites

- node.js ( web tools are made with node first )
- npm ( comes with node )

## Nice-to-haves

- Visual Studio code ( or suitable code editor )
- git ( because it's better than cvs )

# First things first

## Lab repository

<https://github.com/jsoverson/workshop-reverse-engineering>

-or-

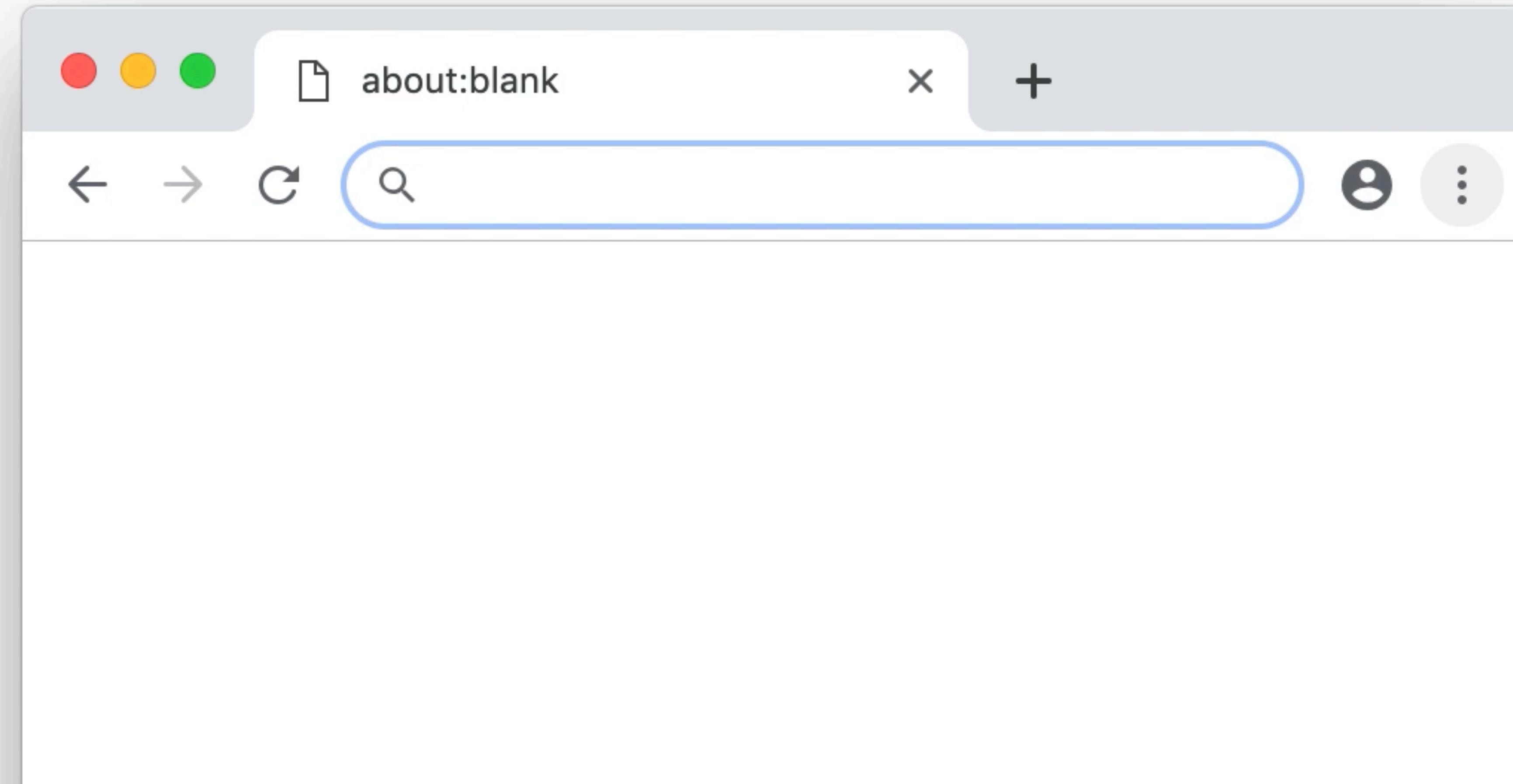
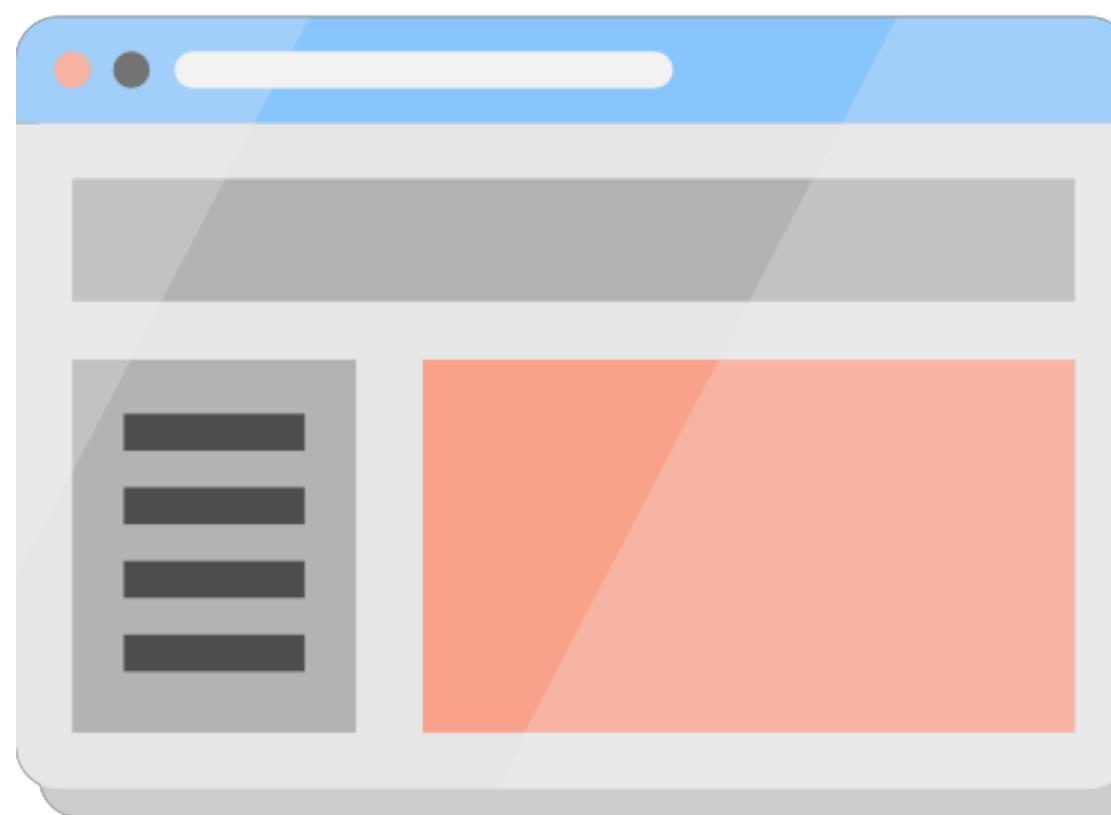
Zipfile: <http://bit.ly/reveng-webapps>

# How to Reverse Engineer Web Applications

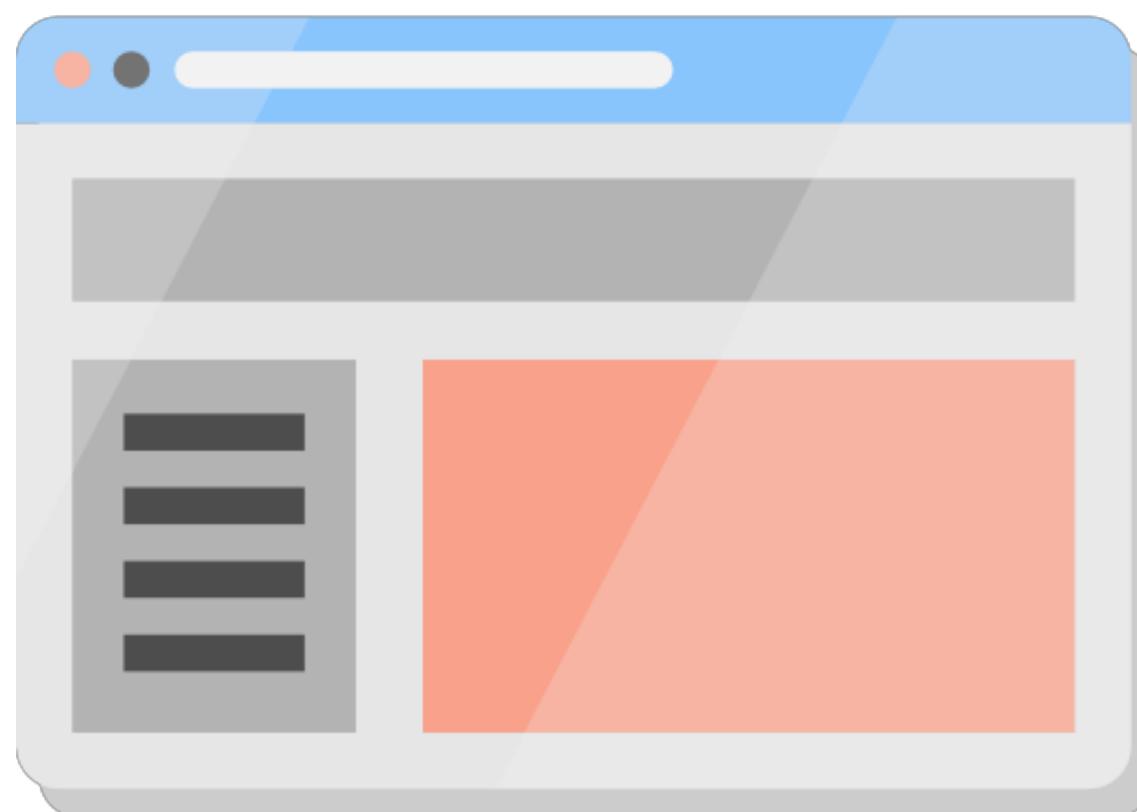
# Challenge #1

1. Web applications have grown extremely complex.

# Web 101



# Web 101



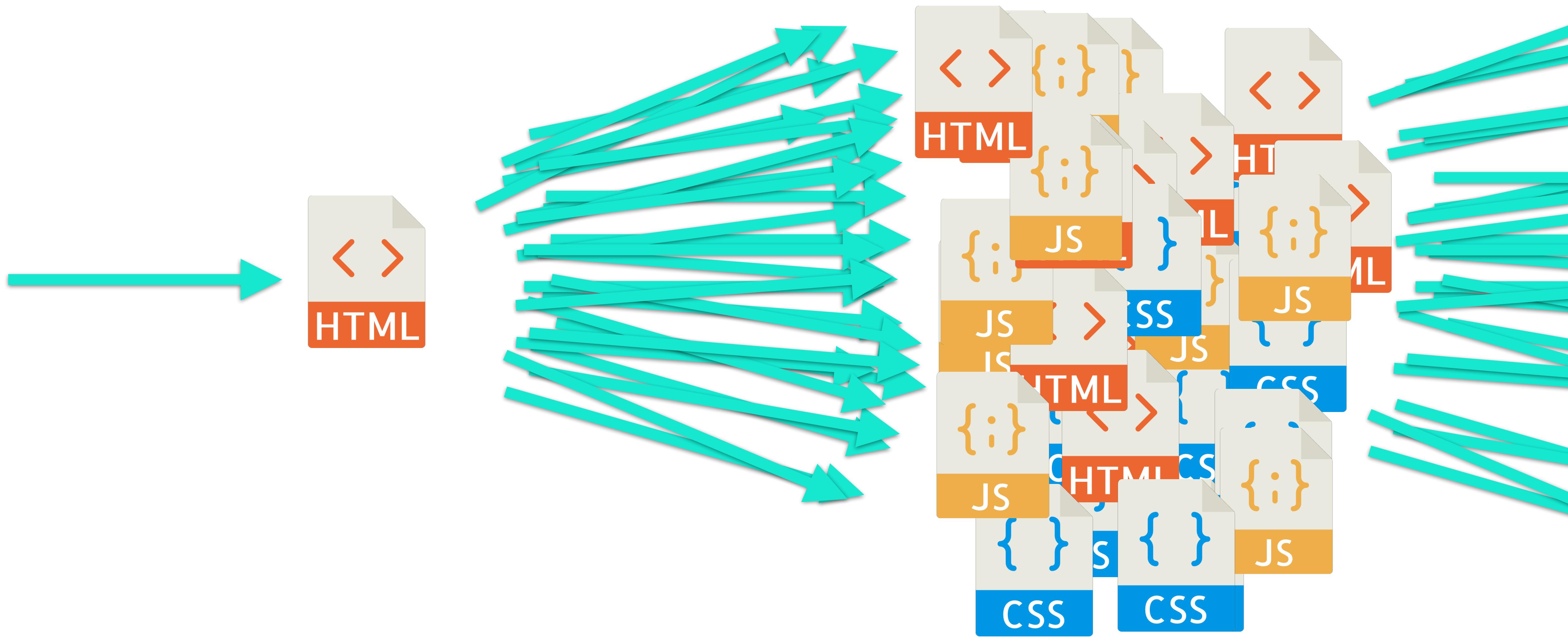
```
GET / HTTP/1.1
Host: www.google.com
Connection: keep-alive
User-Agent: Chrome/72.0.3626.96 Safari/537.36 (...)
```



```
HTTP/1.1 200 OK
Date: Tue, 19 Feb 2019 21:34:08 GMT
Content-Type: text/html
Content-Length: 1932

<!doctype HTML>(...)
```

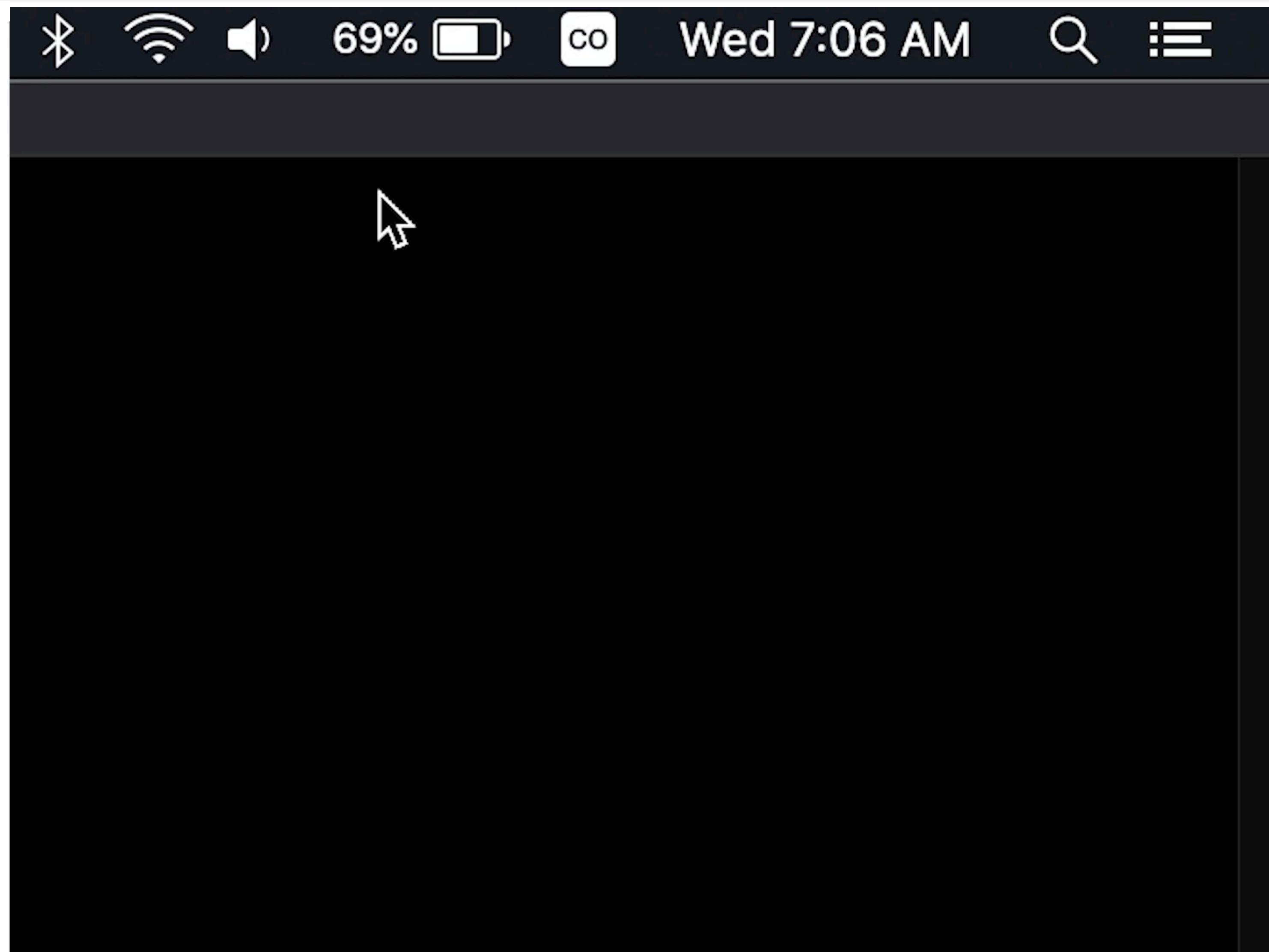
# Web 101



# Challenge #2

1. Web applications have grown extremely complex.
2. Limited ability to run old versions of resources or use old APIs.

99% of websites assume connectivity



# Challenge #3

1. Web applications have grown extremely complex.
2. Limited ability to run old versions of resources or use old APIs.
3. Web site resources can change *frequently*.



# Going to Market Faster: Most Companies Are Deploying Code Weekly, Daily, or Hourly

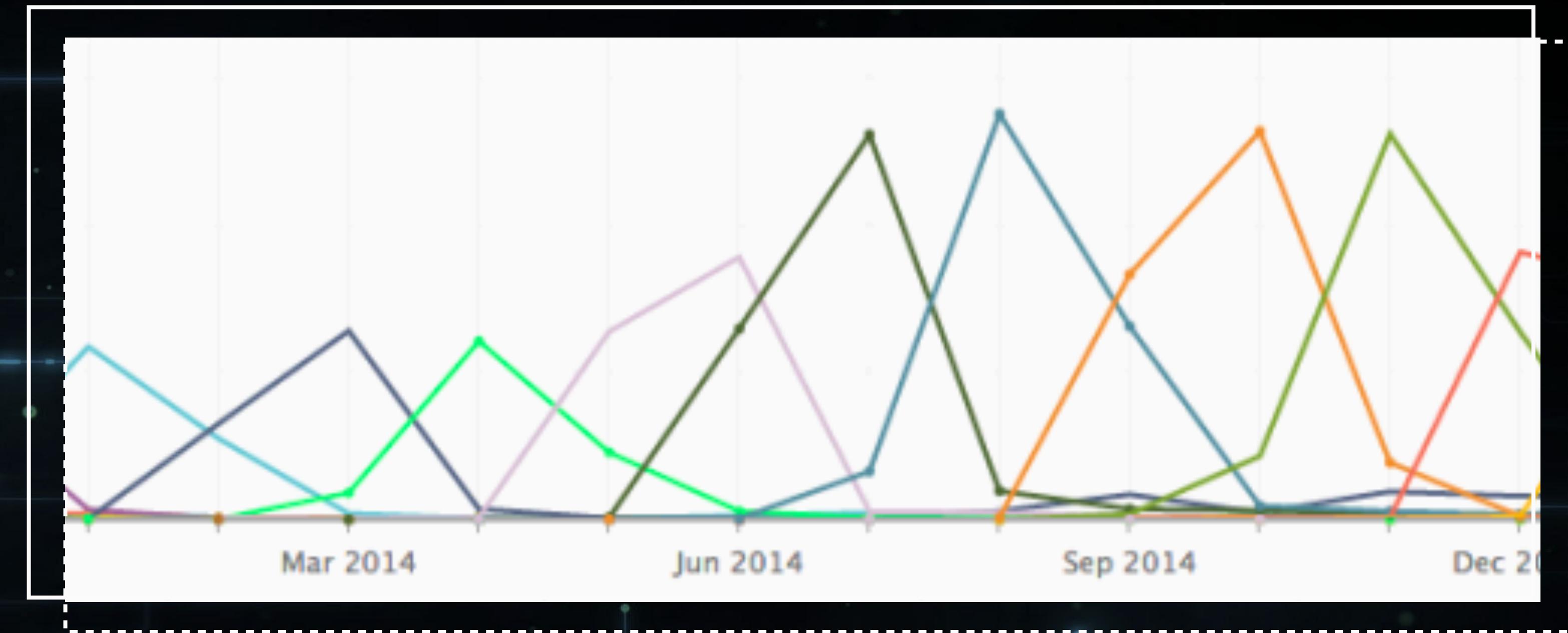
# Etsy Deploy Stats: 2012

- Deployed to production **6,419 times**
- On average, **535/month, 25/day**

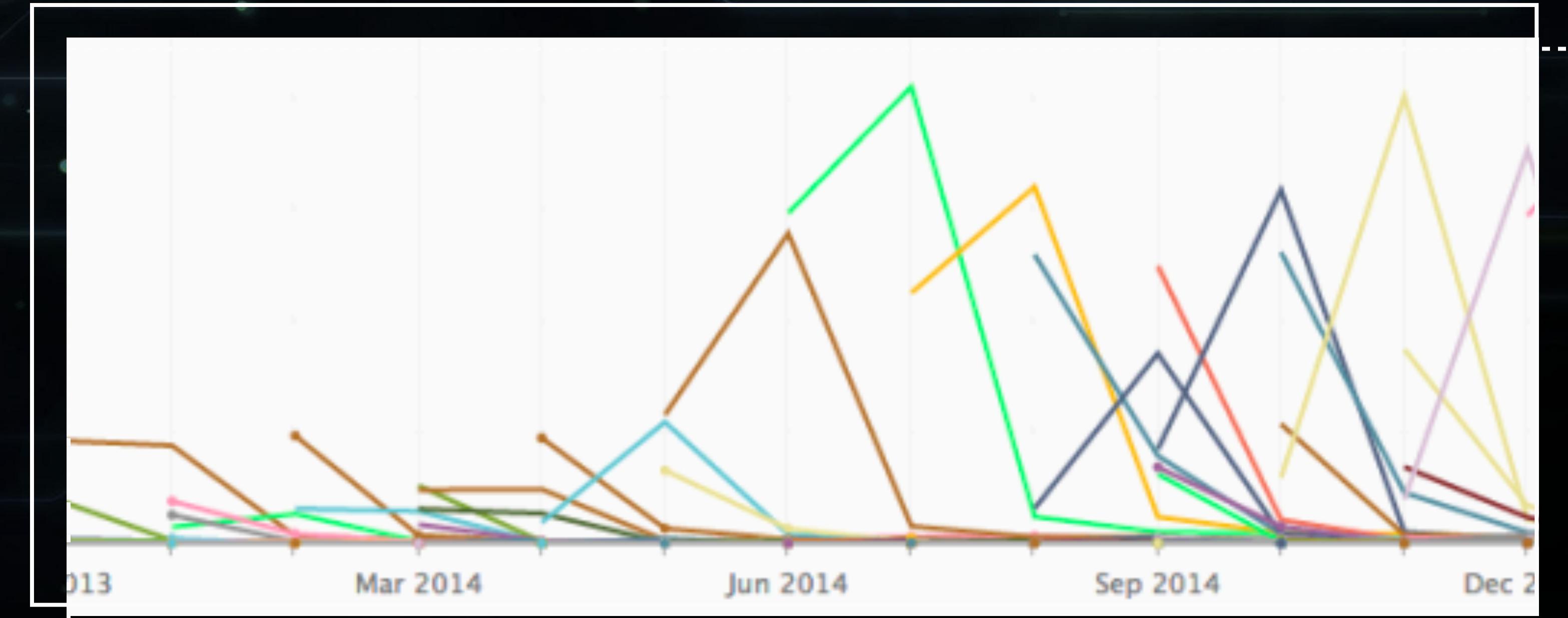
# Challenge #4

1. Web applications have grown extremely complex.
2. Limited ability to run old versions of resources or use old APIs.
3. Web sites and APIs can change *frequently*.
4. Web browsers change *frequently*.

A year of  
Chrome



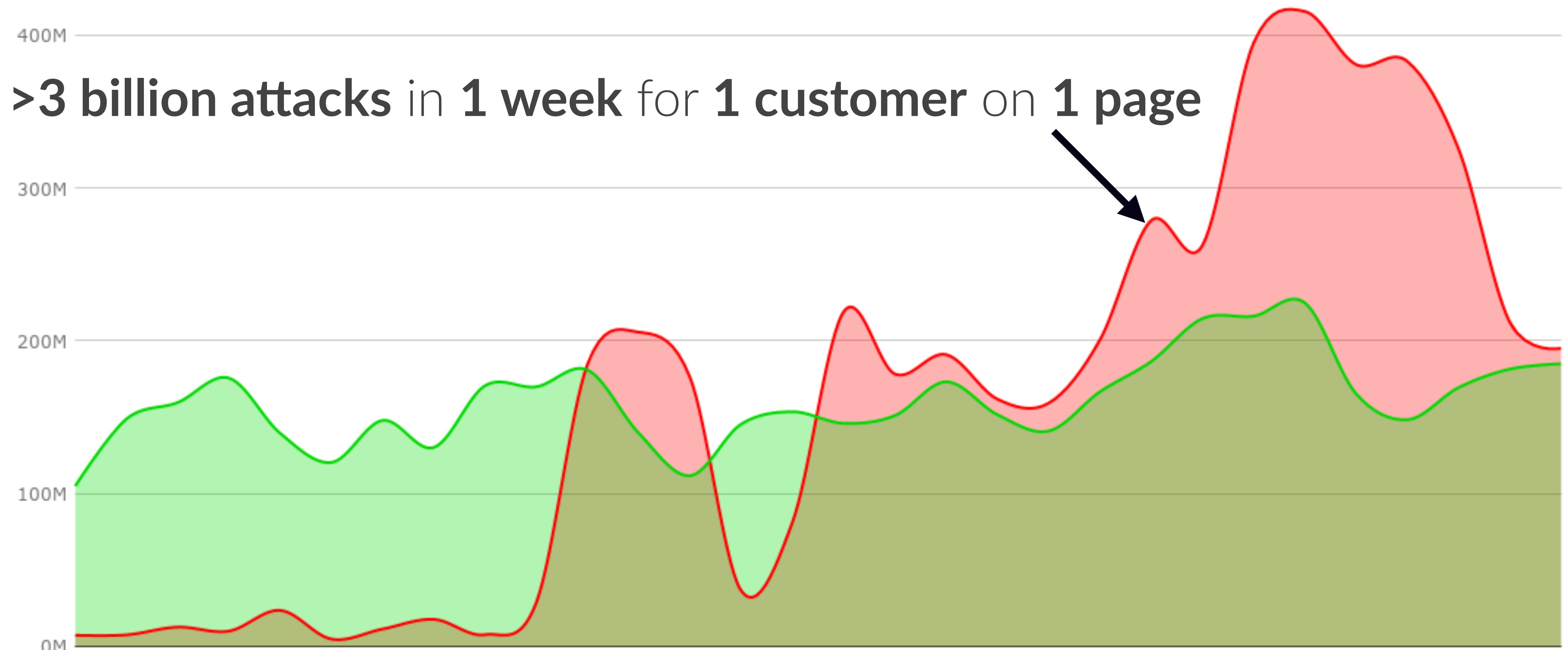
A year of  
FireFox



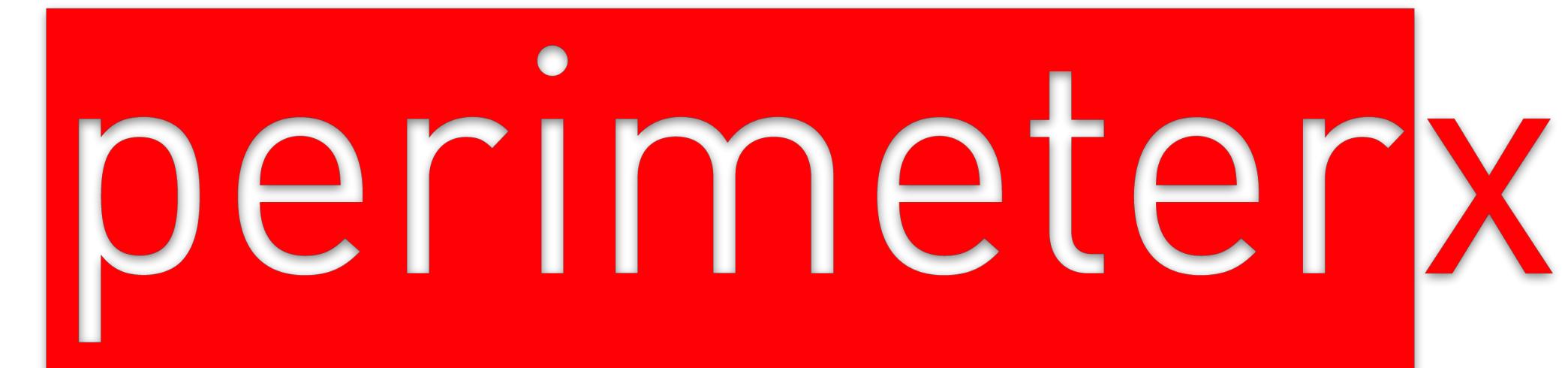
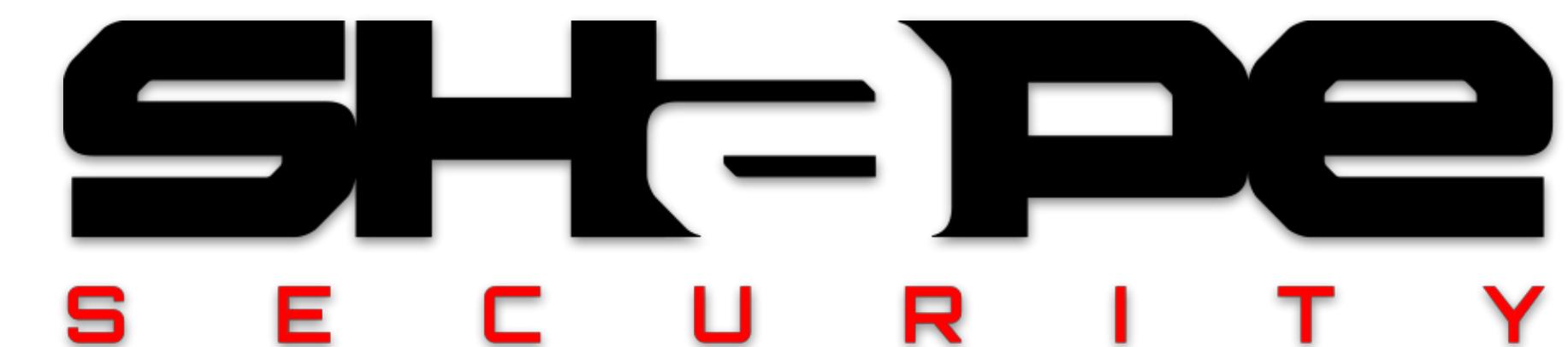
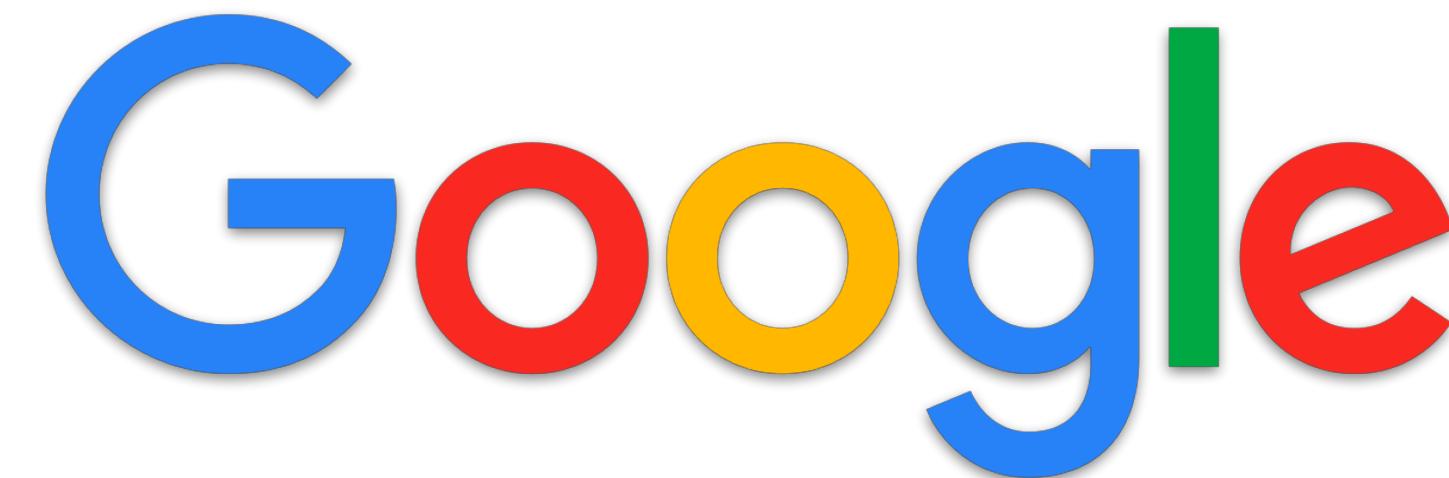
# Challenge #5

1. Web applications have grown extremely complex.
2. Limited ability to run old versions of resources or use old APIs.
3. Web sites and APIs can change *frequently*.
4. Web browsers change *frequently*.
5. Actual attackers are leading to more effective countermeasures.

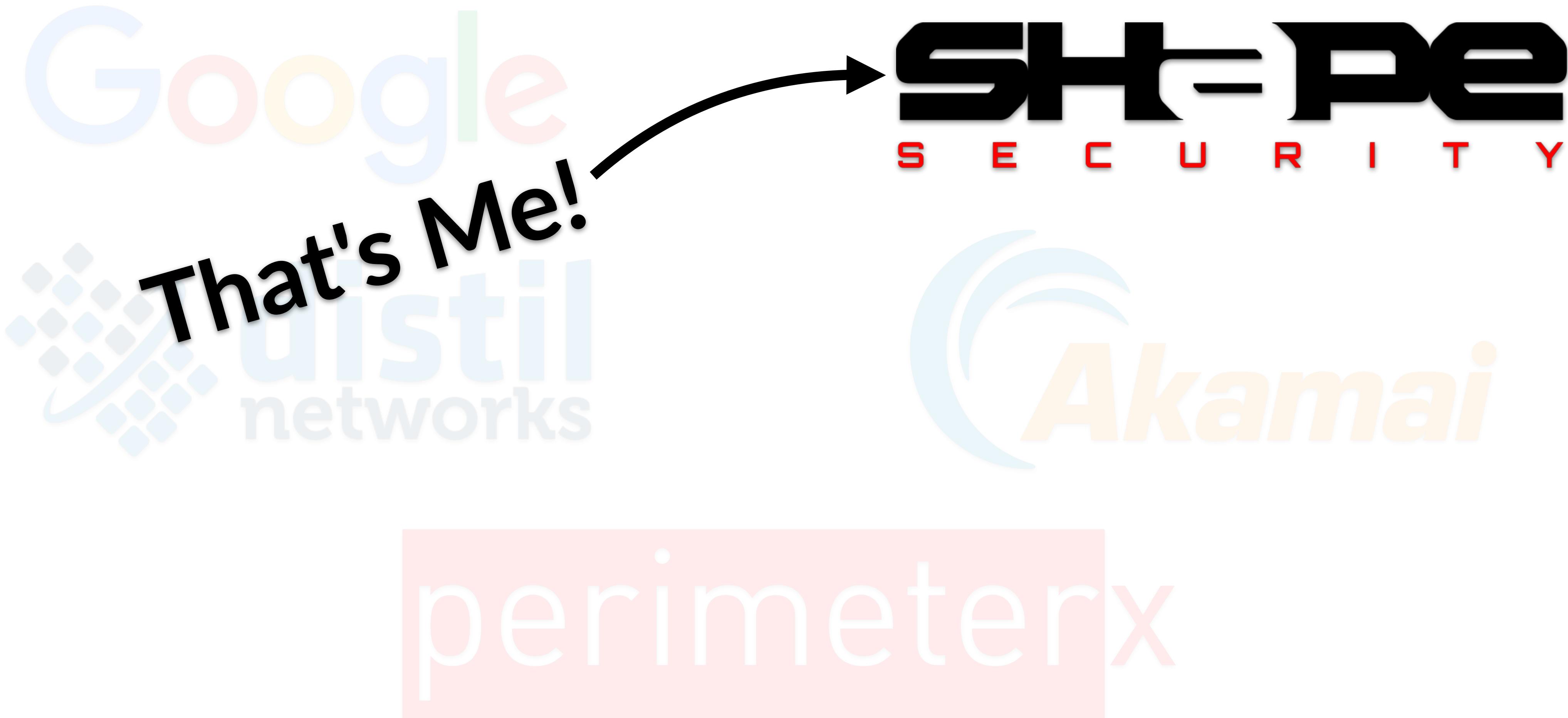
# Website attacks are a serious problem



And have an industry around protecting them



And have an industry around protecting them



So how do you hack web apps?

1. Drive the browser programmatically
2. Simulate and intercept "system" calls
3. Reuse as much application code as possible

# Lab work

## Lab 0

1. Understanding Intent in JavaScript

## Lab 1

1. Extracting logic
2. Extracting logic resiliently
3. Transforming with scope awareness

## Lab 2

1. Automating a browser
2. Intercepting requests
3. Modifying responses
4. Rewriting JS on the fly with a mocked environment

# Lab Format

```
lab-#.#/  
  └── answer  
      └── answer-#.#.js  
  └── test  
      └── test.js  
  └── work  
      └── lab-#.#.js  
└── package.json
```

# Node/npm basics

node [script.js]

npm install

npm install [specific package]

<https://gitpod.io/>



Features   Business   Docs   Blog

Login

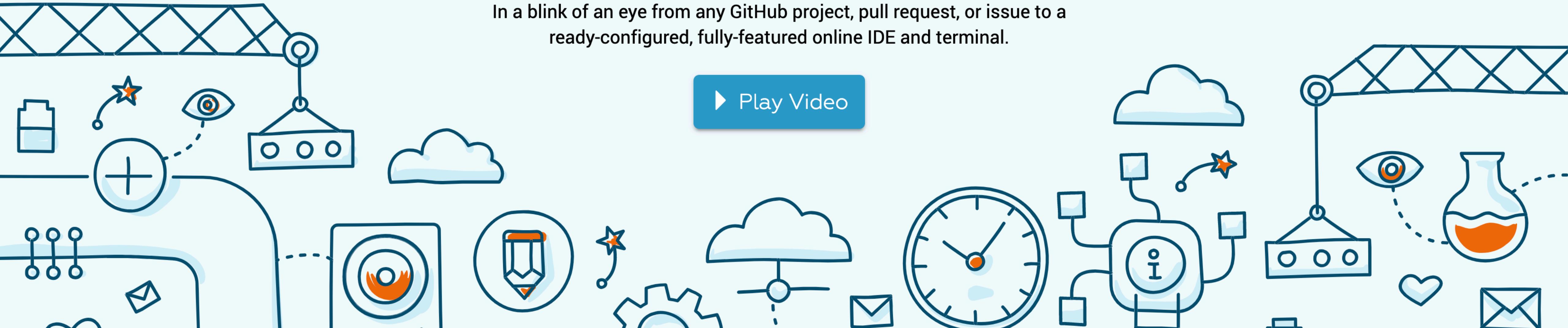
Stay in the flow!

# One-Click Online IDE for GitHub

and other code hosting platforms.

In a blink of an eye from any GitHub project, pull request, or issue to a ready-configured, fully-featured online IDE and terminal.

▶ Play Video



# <https://try-puppeteer.appspot.com/>

Try Puppeteer v1.9.0

Run an example: [screenshot.js](#)



```
1 const browser = await puppeteer.launch();
2
3 const page = await browser.newPage();
4 await page.goto('https://example.com');
5
6 console.log(await page.content());
7 await page.screenshot({path: 'screenshot.png'});
8
9 await browser.close();
10
```

RUN IT

LOG



# Lab 0.1

Understanding Intent in JavaScript

## Background

Payload A is the first script of three found in an exploited dependency of npm package event-stream.

## Goal

De-obfuscate payload-A.js and identify how it is loading the second payload.

# Lab 0.1

## Understanding Intent in JavaScript

### Tips

- Format your code with `⌘+⬆️+P` or `^+⬆️+P` to open command palette then "Format Document"
- Rename variables by pressing F2 when cursor is over an identifier.
- There is a helper file that includes additional encoded strings.  
(The first and second strings in the helper file are beyond the scope of this lab.)
- `process.env` contains the environment variables at time of execution

# Lab Series 1

Programmatically manipulating JavaScript

# Lab 1.1

Programmatically extract logic from JavaScript

## Background

Common JavaScript best practices and bundlers produce code that has a minimal public footprint, limiting the ability to hook into existing logic.

## Goal

Use the Shift parser and code generator to **read payload-A.js** and **extract its e function** and **export it as an unhex method** in our node script.

# Lab 1.1

Using parsers

`parse(originalSource) → Abstract Syntax Tree (AST)`

`codegen(AST) → newSource`

# Lab 1.1

What is an AST?

```
const msg = "Hello World";
```



VariableDeclarationStatement

# Lab 1.1

What is an AST?

```
const msg = "Hello World";
```



VariableDeclaration

# Lab 1.1

What is an AST?

```
const msg = "Hello World";
```



VariableDeclarator

# Lab 1.1

What is an AST?

```
const msg = "Hello World";
```

A diagram illustrating the structure of the code. It features a horizontal line with three curly braces underneath. The first brace covers the word 'msg'. The second brace covers the string 'Hello World'. The third brace is positioned below the second one, indicating they are part of the same expression.

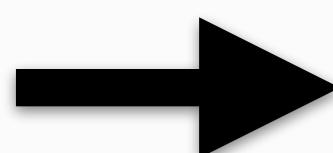
BindingIdentifier

LiteralStringExpression

# Lab 1.1

What is an AST?

```
1 const msg = "Hello World";
2
3 log(msg); |
```

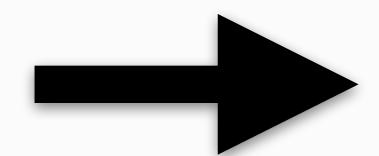


```
1 ▾ {  
2   "type": "Script",  
3   "directives": [],  
4   "statements": [  
5     {  
6       "type": "VariableDeclarationStatement",  
7       "declaration": {  
8         "type": "VariableDeclaration",  
9         "kind": "const",  
10        "declarators": [  
11          {  
12            "type": "VariableDeclarator",  
13            "binding": {  
14              "type": "BindingIdentifier",  
15              "name": "msg"  
16            },  
17            "init": {  
18              "type": "LiteralStringExpression",  
19              "value": "Hello World"  
20            }  
21          }  
22        ]  
23      }  
24    }
```

# Lab 1.1

What is an AST?

```
{  
  "type": "Script",  
  "directives": [],  
  "statements": [  
    {  
      "type": "VariableDeclarationStatement",  
      "declaration": {  
        "type": "VariableDeclaration",  
        "kind": "const",  
        "declarators": [  
          {  
            "type": "VariableDeclarator",  
            "binding": {  
              "type": "BindingIdentifier",  
              "name": "msg"  
            },  
            "init": {  
              "type": "LiteralStringExpression",  
              "value": "Hello World"  
            }  
          }  
        ]  
      }  
    ]  
  }  
}  
  
"Hello WOPRs"
```



```
1 const msg = "Hello WOPRs";  
2  
3 log(msg);
```

# Lab 1.1

Programmatically extract logic from JavaScript

## Background

Common JavaScript best practices and bundlers produce code that has a minimal public footprint, limiting the ability to hook into existing logic.

## Goal

Use the Shift parser and code generator to **read payload-A.js** and **extract its e function** and **export it as an unhex method** in our node script.

# Lab 1.1

Programmatically extract logic from JavaScript

## Tips

- Don't overthink it - you're just deeply accessing a property in an object.
- `console.log()` as you make your way down the tree, e.g.  
`console.log(tree.expressions[0]);`
- Copy/paste `payload-A.js` into [astexplorer.net](https://astexplorer.net) for an interactive UI

# Lab 1.2

Resiliently extract logic from JavaScript

## Background

Extracting and manipulating JavaScript requires that the code be resilient to changes in the input source.

## Goal

Use a traversal method to pick out the same function from lab 1.1 by inspecting the AST node of the function and identifying it by its shape or attributes.

# Lab 1.2

Resiliently extract logic from JavaScript

## Tips

- `allNodes` contains a list of all nodes in the AST.
- You can iterate over that list and check every node for properties that represent the node you want to target.
- You probably want to check if `node.type === 'FunctionDeclaration'`
- You can then check the function name to make sure it is equal to '`e`'

# Lab 1.3

Rewrite JavaScript taking scope and context into account

## Background

Rewriting JavaScript requires tools that are aware of the entire program's scope and context.

## Goal

Use **shift-scope** to rename global variables "a" and "b" to "first" and "second"

# Analyzing Scope

```
1
2  /* rename a to first and b to second */
3  const a = "Hello";
4  const b = "World";
5
6  function join(a, b) {
7    /* these should not be renamed */
8    return a + b;
9  }
10
11 module.exports = function() {
12   /* these should be renamed */
13   return join(a, b);
14 }
15
```

# Analyzing Scope

```
const scope = analyzeScope(AST)
```

# Analyzing Scope

- Scope
  - `children` Child scopes
  - `type` Global / Script / Function etc
  - `astNode` The root node
  - `variableList` The variables declared or referenced in this scope

# Analyzing Scope

```
const lookupTable = new ScopeLookup(scope)
```

```
const identifier = /* node from AST */
```

```
const lookup = lookupTable.variableMap.get(identifier)
```

# Lab 1.3

Rewrite JavaScript taking scope and context into account

## Background

Rewriting JavaScript requires tools that are aware of the entire program's scope and context.

## Goal

Use **shift-scope** to rename global variables "a" and "b" to "first" and "second"

# Lab 1.3

Rewrite JavaScript taking scope and context into account

## Tips

- The argument to `lookupTable.variableMap.get()` should be the identifier node itself, not a string.
- There are variables already defined that reference the AST nodes.
- Each lookup returns a list of entries with declarations and references. You need to change *both* declarations and references to fully rename an identifier.

# Lab Series 2

Programmatically controlling a browser

# Lab Series 2

## **Notes:**

The Puppeteer npm package downloads Chrome on every install. If internet is flakey, download it once and copy `node_modules/puppeteer` from one lab to another

# Lab 2.1

Programmatically control a browser with Puppeteer

## Background

Web application analysis needs to be completely automated, otherwise everything depending on a manual step risks breaking when anything changes.

## Goal

Set up a base environment that controls Chrome with Puppeteer and nodejs.

# What is Puppeteer?

Puppeteer is a nodejs library that provides a high-level API to control Chrome over the [DevTools Protocol](#).

Puppeteer runs [headless](#) by default, but can be configured to run full (non-headless) Chrome or Chromium.

# Puppeteer API

```
puppeteer.launch(); → browser  
puppeteer.connect();
```

# Browser instance

```
browser.pages();  
browser.newPage();
```

→ page  
( Tab )

# Page instance

```
page.goto();  
page.evaluate();
```

```
page.$();  
page.$$();
```



element

# Element instance

```
element.click();  
element.type(...);  
element.tap();
```

# Lab 2.1

Programmatically control a browser with Puppeteer

## Tips

- The Puppeteer API is fantastic : <http://bit.ly/puppeteer-api>
- You need to get a **browser** instance from **puppeteer**
- You need to get a **page** instance from **browser**
- You need to go to a url of your choice, e.g. <https://example.com>

# Lab 2.2

Intercept requests via the Chrome Devtools Protocol

## Background

Intercepting, inspecting, and modifying inbound and outbound communication is a critical part of any reverse engineering effort.

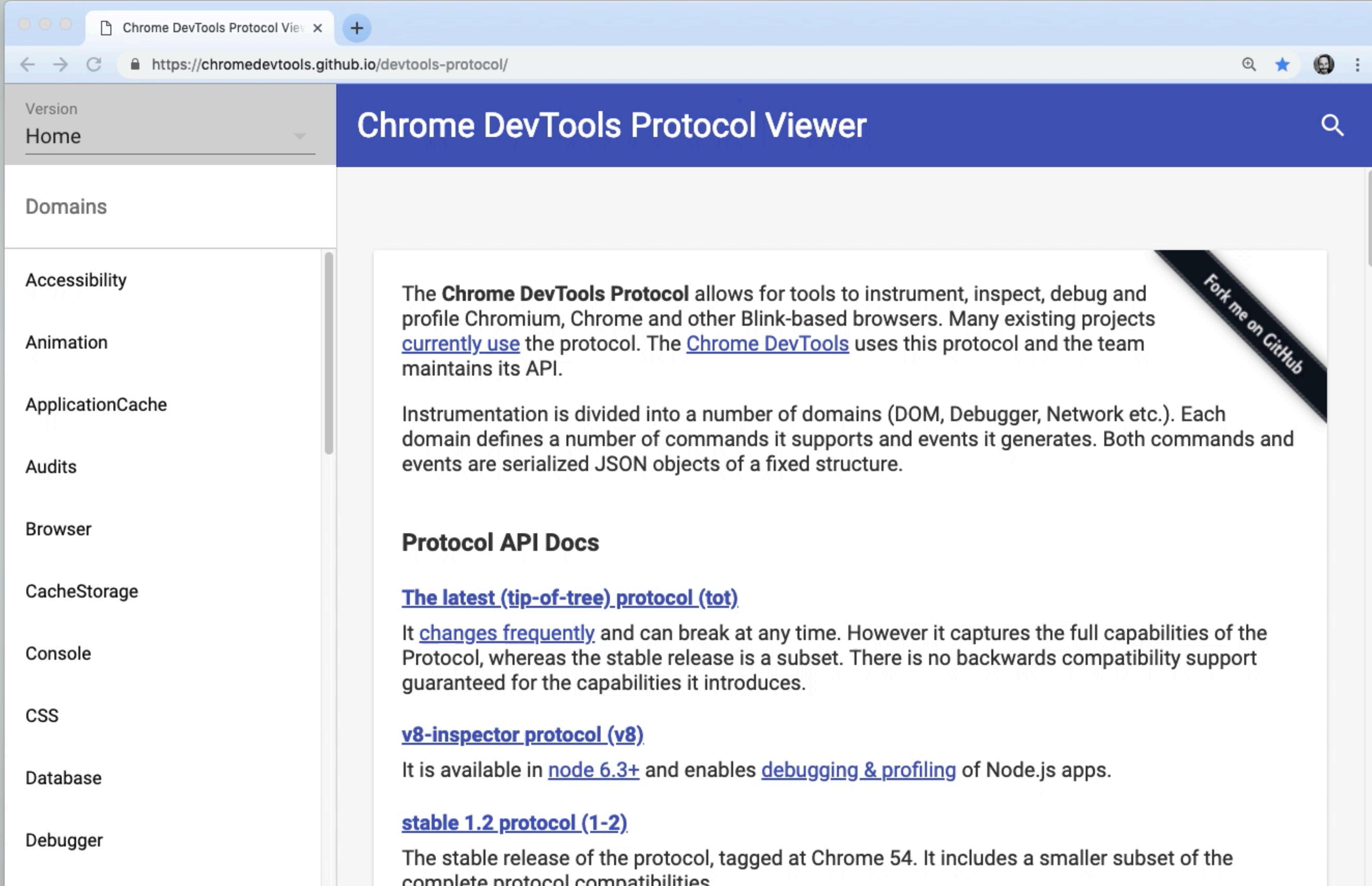
## Goal

Use the Chrome Devtools Protocol directly to intercept all Script resources, log the URL to the terminal, and then continue the request.

# What is Chrome Devtools Protocol?

The Chrome DevTools Protocol allows for tools to instrument, inspect, debug and profile Chromium, Chrome and other Blink-based browsers.

<http://bit.ly/chrome-devtools-protocol>



The screenshot shows a web browser window displaying the "Chrome DevTools Protocol Viewer" at <https://chromedevtools.github.io/devtools-protocol/>. The page has a blue header with the title "Chrome DevTools Protocol Viewer". On the left, a sidebar lists various domains: Version, Home, Domains, Accessibility, Animation, ApplicationCache, Audits, Browser, CacheStorage, Console, CSS, Database, and Debugger. The main content area contains text about the protocol, links to API documentation, and information about stable releases.

The **Chrome DevTools Protocol** allows for tools to instrument, inspect, debug and profile Chromium, Chrome and other Blink-based browsers. Many existing projects [currently use](#) the protocol. The [Chrome DevTools](#) uses this protocol and the team maintains its API.

Instrumentation is divided into a number of domains (DOM, Debugger, Network etc.). Each domain defines a number of commands it supports and events it generates. Both commands and events are serialized JSON objects of a fixed structure.

## Protocol API Docs

[The latest \(tip-of-tree\) protocol \(tot\)](#)

It [changes frequently](#) and can break at any time. However it captures the full capabilities of the Protocol, whereas the stable release is a subset. There is no backwards compatibility support guaranteed for the capabilities it introduces.

[v8-inspector protocol \(v8\)](#)

It is available in [node 6.3+](#) and enables [debugging & profiling](#) of Node.js apps.

[stable 1.2 protocol \(1-2\)](#)

The stable release of the protocol, tagged at Chrome 54. It includes a smaller subset of the complete protocol compatibilities.

*Fork me on GitHub*

# Initiating a CDP Session

```
const client = page.target().createCDPSession();
```

# Sending commands

```
client.send("command");  
client.send("command", {options...});  
client.on("event", listener);
```

# Intercepting Network Traffic

(pseudo code)

```
send("Network.enable");
send("Network.setRequestInterception", patterns);
on("Network.requestIntercepted", (evt) => {
    /* modify request or response */

    send(
        "Network.continueInterceptedRequest",
        request
    )
})
```

# Lab 2.2

Intercept requests via the Chrome Devtools Protocol

## Background

Intercepting, inspecting, and modifying inbound and outbound communication is a critical part of any reverse engineering effort.

## Goal

Use the Chrome Devtools Protocol directly to intercept all Script resources, log the URL to the terminal, and then continue the request.

# Lab 2.2

Intercept requests via the Chrome Devtools Protocol

## Tips

- The Puppeteer API is fantastic : <http://bit.ly/puppeteer-api>
- This is purely in the **Network** domain
- Remind me to flip back to the pseudo-code if you're stuck.

# Lab 2.3

Modify intercepted requests

## Background

Modifying intercepted requests requires recreating an entire HTTP response and passing it along as the original.

## Goal

Retrieve the original script body and append a `console.log()` statement to the end of the script that simply logs a message.

# Intercepting Network Traffic

(pseudo code)

```
send("Network.enable");
send("Network.setRequestInterception", patterns);
on("Network.requestIntercepted", (evt) => {
    const response =
        send("Network.getResponseBodyForInterception", interceptionId);

    send(
        "Network.continueInterceptedRequest",
        request
    )
})
```

# Modifying a Response

(pseudo code)

```
const response =  
  send('Network.getResponseBodyForInterception', interceptionId);  
  
const body = response.base64Encoded  
  ? atob(response.body)  
  : response.body;  
  
send('Network.continueInterceptedRequest', {  
  interceptionId,  
  rawResponse: btoa(/* Complete HTTP Response */)  
});
```

# HTTP Requests & Responses

## Requests

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; ... )... Firefox/51.0
Accept: text/html,application/xhtml+xml,...,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345
-12656974
(more data)
```

## Responses

```
HTTP/1.1 403 Forbidden
Server: Apache
Content-Type: text/html; charset=iso-8859-1
Date: Wed, 10 Aug 2016 09:23:25 GMT
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
Age: 3464
Date: Wed, 10 Aug 2016 09:46:25 GMT
X-Cache-Info: caching
Content-Length: 220
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
(more data)
```

start-line

HTTP headers

empty line

body

# Lab 2.3

Modify intercepted requests

## Background

Modifying intercepted requests requires recreating an entire HTTP response and passing it along as the original.

## Goal

Retrieve the original script body and append a `console.log()` statement to the end of the script that simply logs a message.

# Lab 2.3

Modify intercepted requests

## Tips

- Rely on the Chrome Devtools Protocol documentation : <https://bit.ly/chrome-devtools-protocol>
- What you add to each script is up to you, it's user choice as long as it is observable.
- The last TODO requires reading the CDP documentation.

# Final Lab

Meaningfully rewrite a script to intercept its access to browser APIs

## Background

Intercepting a script's access to standard APIs allows you to guide its execution in the direction you want without modifying its internals.

## Goal

Wrap the example site's script to intercept access to `document.location` to make the script think it is hosted elsewhere & inject code that exposes the seed