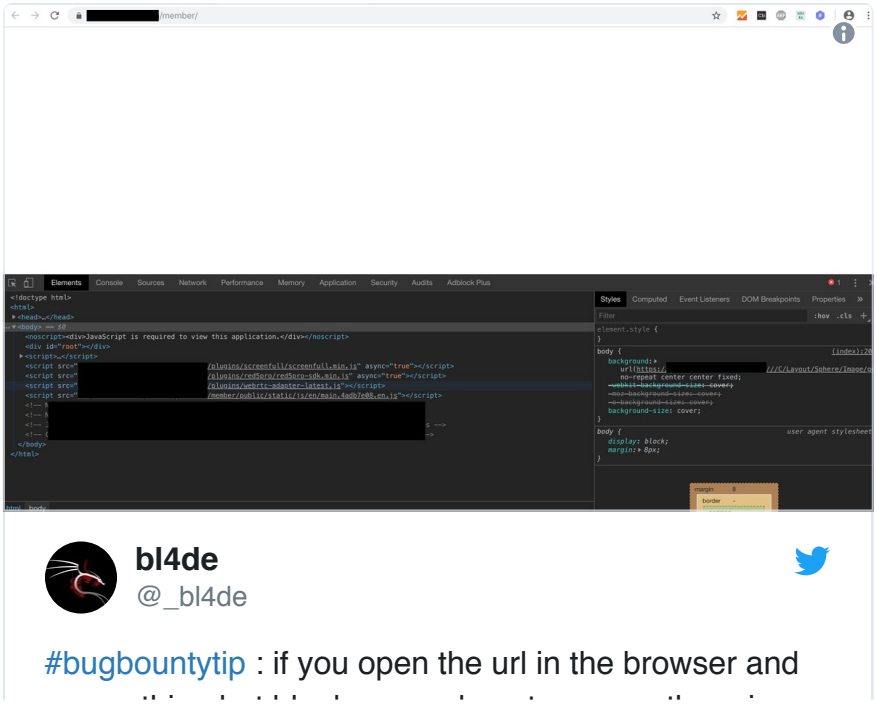
How to perform the static analysis of website source code with the browser — the beginner's bug bounty hunters guide

bl4de Follow
Nov 1 · 20 min read

In this guide I am going to show you how to use web browser built-in tools to investigate web application clientside source code. This might sounds weird, as probably browser is not the best choice to perform such task, but before you will go deeper and start intercepting HTTP requests using Burp Suite or poking around with *alert(1)* for endless XSSes here and there, it is always a good idea to know your target well first.

This post is intended mostly for beginner bug bounty hunters with no or little experience in analysing HTML and JavaScript code, but I hope more experienced hackers will find something interesting here as well :)

The idea of this post was born in my head when one of my recent tweets with some really basic tip gained a lot of attention across the community:



As this simple clue is actually a tip of an iceberg, instead of posting similar tips on Twitter where they can be easily missed, I've decided to collect a couple of them into one blog post. I hope some of you will find it useful.

So, let's get started then.

Toolset

Every modern web browser today has a set of developer tools build in. To enable them, you can use Ctrl+Shift+I, CMD+Option+I (macOS), F12 key or simply find the right option in browser's menu—it depends on operating system and browser you are using. Although in this post I will use latest version of Chromium, there is no major differences (besdies UI) if you use Firefox, Safari, Chrome or Edge. It's your choice, but I found Chrome Developer Tools the most powerful (Chrome Developer Tools or simply DevTools are available in Chrome, Chromium, Brave, Opera or any other Chromium based browser).

Another tool which is good to have installed is an IDE (Integrated Development Environment) or any code editor with HTML and JavaScript syntax highlighting. Again, this is all based on your preferences, but I found Visual Studio Code is the best you can get

right now (by the way, VSCode is my one and only IDE I use for literally everything, including my full time job). You can download VSCode for your operating system at this url: https://code.visualstudio.com/

A good idea also is to install NodeJS (and become familiar with using it—there're literally thousands of great resources in the internet). It's available at https://nodejs.org/en/

Python interpreter is the next must-to-have thing for me (chances are if you are using *NIX based operation system, it's already installed. If you're Windows user, you have to install Python on your own). An ability to code in Python is priceless and I recommend everyone to give it a try even if you have never wrote a single line of code in any programming language.

NodeJS is very useful to run and test JavaScript code with the terminal (the same you can achieve using the browser, but we will get to this later with some pros and cons). Python is useful for creating your own scripts like tools, quick Proof of Concepts and actual exploits—I will present some of my own tools later in this post as well. If you feel more familiar with other interpreted language (think of Ruby, PHP, Perl, Bash etc.) you can use them as well if you wish. The main advantage of such languages is that scripts can be run without compilation, right from the command line, they are 100% portable between different platforms and they have a lot of features built in thousands of libraries or modules available all over the web.

Ok, it's finally time to get our hands dirty.

Investigating HTML source code

Let's get back for a moment to my tweet I've quoted earlier. As you might notice, the screenshot presents a website which looks like there is no content there, only white, blank page.

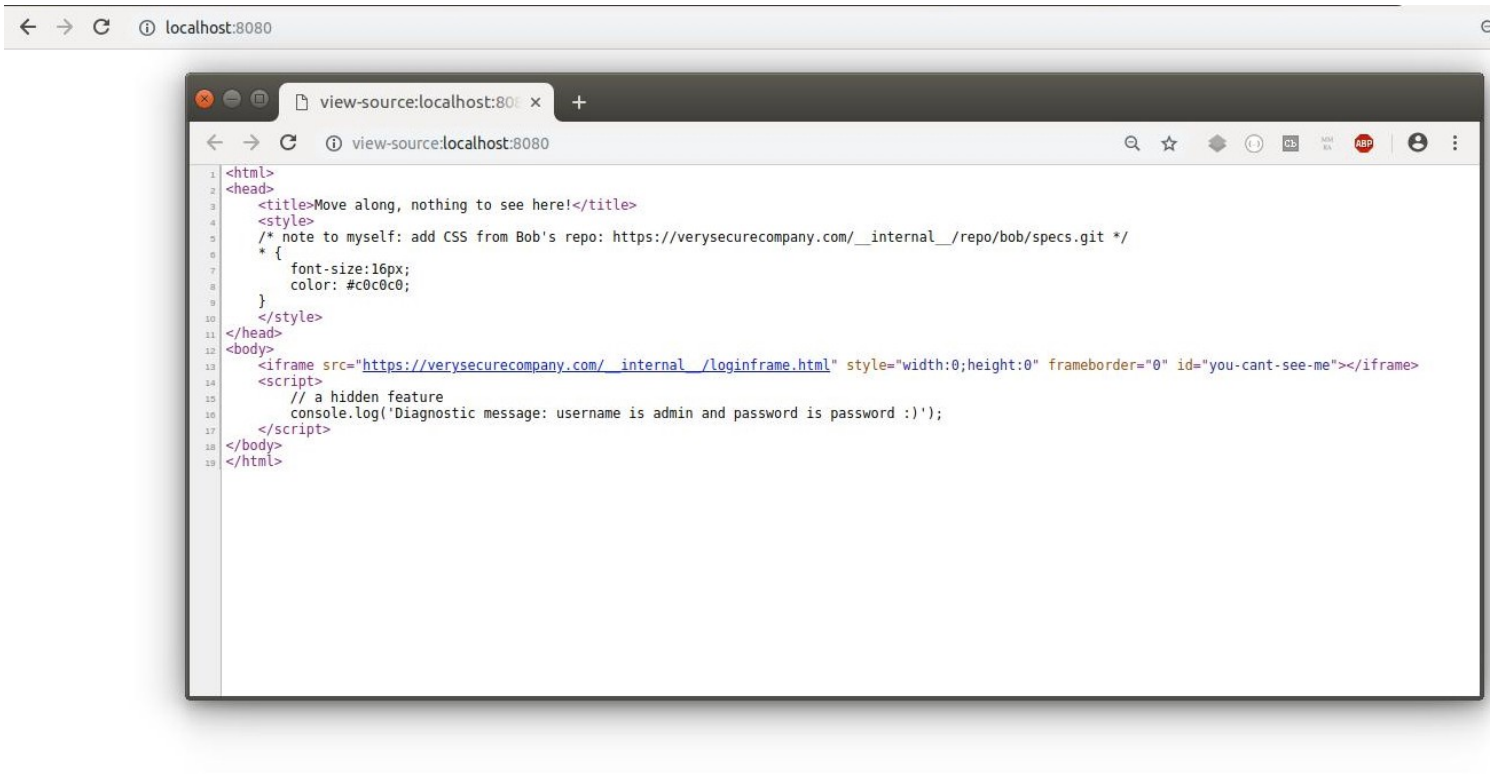
But if you take a look at source code (using CTRL+U or CMD+Option+U on macOS) you can see a plenty of code (unfortunately I can't provide an url for the website from the screenshot as it comes from private bug bounty program). Why those elements can't be seen in the browser?

Important thing here is that **some HTML tags do not render any content to the page**. There are couple of them, but most common are *<html>, <head>, <body>, <style>* or*<script>*. Also, CSS allows to hide elements (e.g. by setting their *width* and *height* CSS property to 0 or setting *display* to *none).*

Consider following example:

```
<html>
<head>
    <title>Move along, nothing to see here!</title>
    <style>
    /* note to myself: add CSS from Bob's repo: https
    * {
        font-size:16px;
        color: #c0c0c0;
    }
    </style>
</head>
<body>
    <iframe src="https://verysecurecompany.com/__inte
```

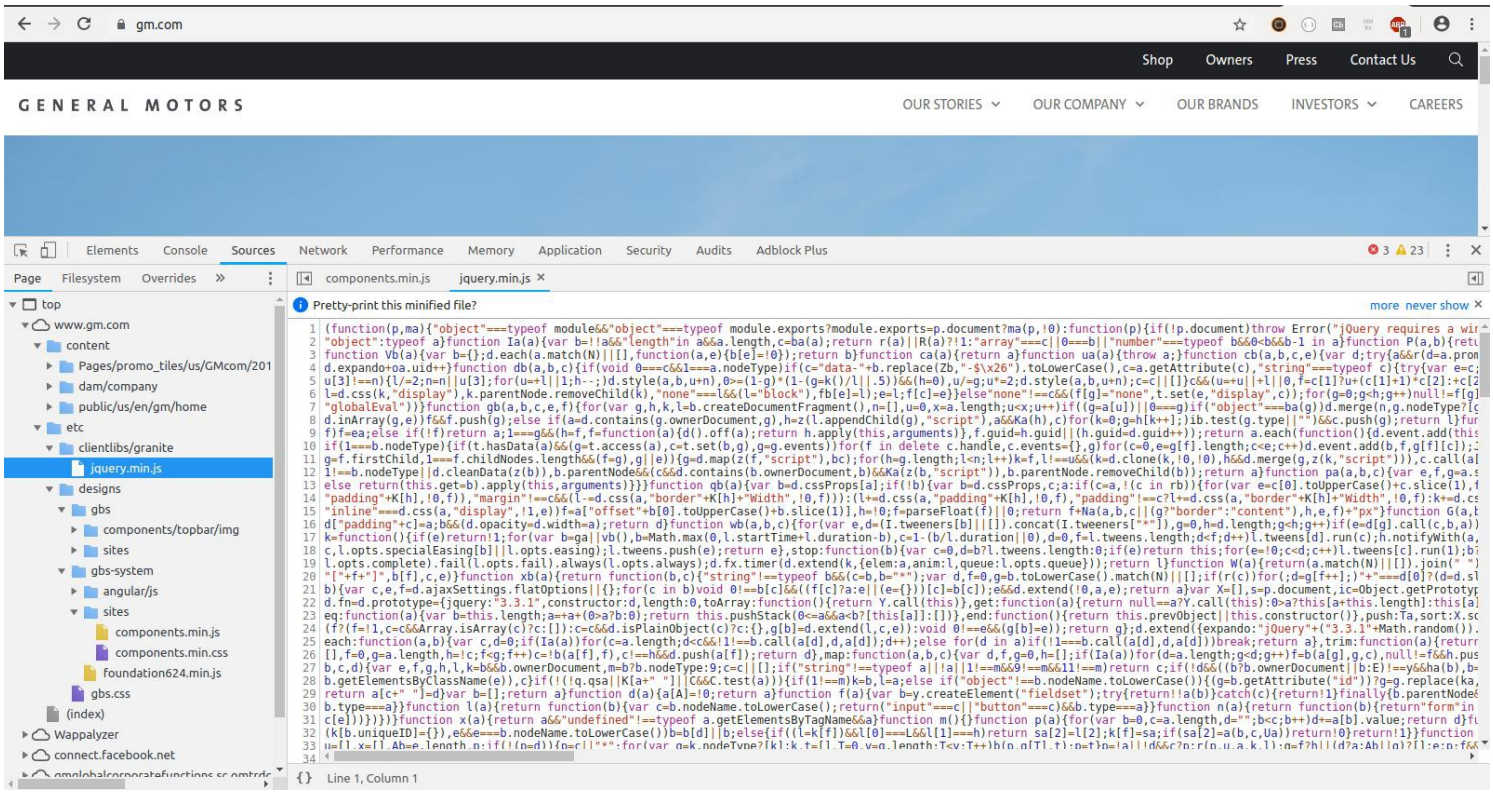
If you open this simple HTML page in the browser, it won't render any content and you won't see anything from what's in it. But when you take a look at source code, things are going to become much more interesting:



Using CTRL+U or CMD+Option+U (macOS) you can see website source code.

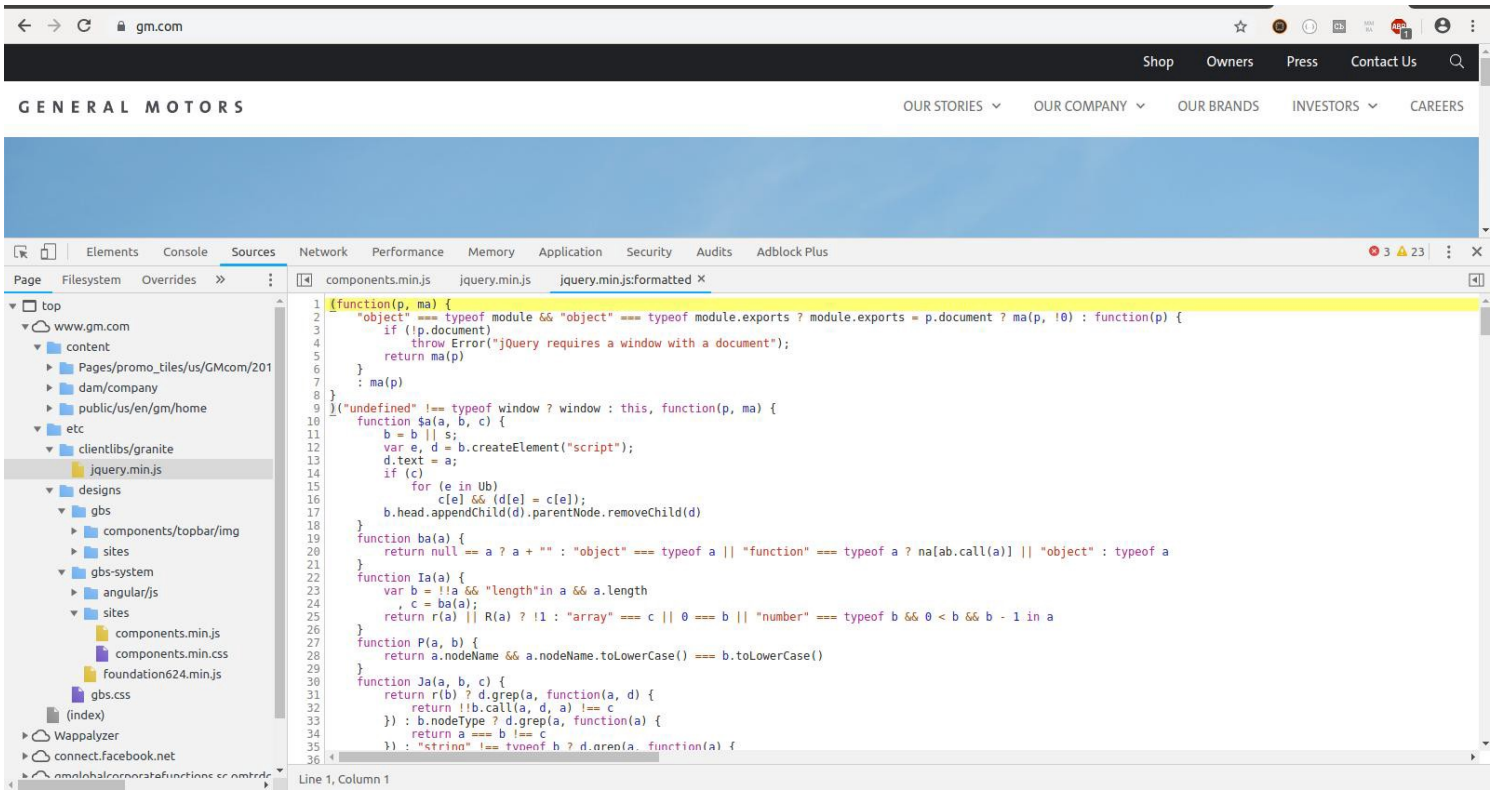
There are so many valuable information: urls to internal resources, "hidden" frame with login form and even a diagnostic message with some credentials which will be printed in developer tools console. None of those elements are visible on the page. Of course, do not expect you will find such information on every single website, however commented fragments of JavaScript code are very common and sometimes they reveal API endpoints which are still accessible on the server side of the app.

But using *View source* option doesn't give you the whole picture, because **it presents only the current HTML document**. What's much more interesting are all additional resources loaded by *<iframe>*,*<script>* or similar tags. You can see those resources in *Sources* tab in Chrome Developer Tools:



Sources tab allows you to see all resources loaded by the website

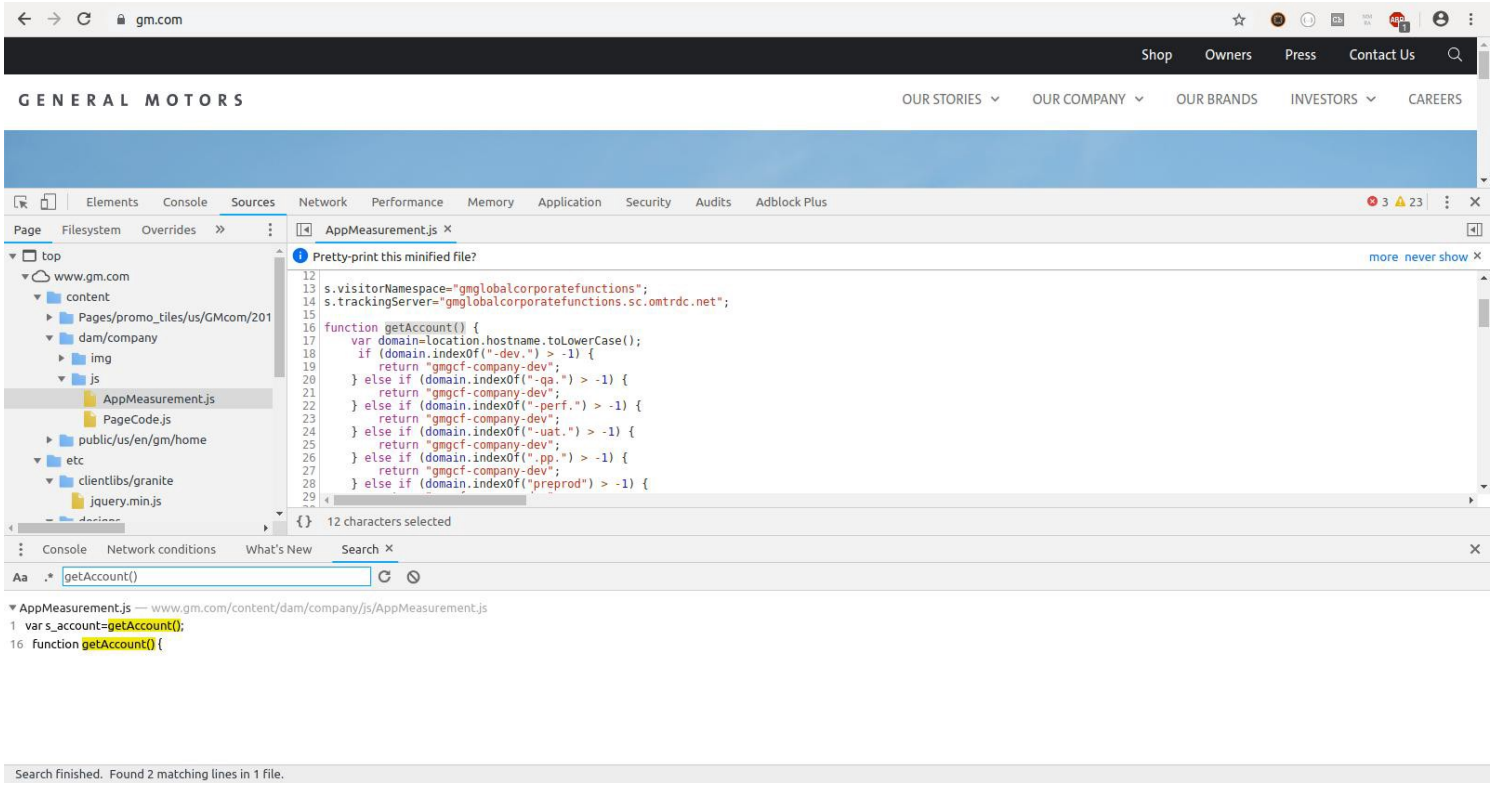
(index) node at the very bottom of the tree is the main HTML document you are able to see with *View source* option. All other resources are presented as a standard folders and files tree. If you click any of those files, you will see its content on the right. In the screenshot presented, it's *jquery.min.js* file and it's a common that you will find minified version of all JavaScript files (this is a good practice from web application performance point of view). But if you click the small icon *{}* at the bottom, DevTools will "unminify" the code for you and makes it much easier to read:



jquery.min.js:formatted tab presents unminified JavaScript file from the tab on the left

Some of the websites uses a special feature known as source map (source maps are used to map minified functions, variables and objects names into their "real" names as well as position in original source code—you can find more about source maps at this url—https://developers.google.com/web/tools/chrome-devtools/javascript/source-maps). Source maps make formatted code even more readable by providing meaningful naming of all objects, instead of identifiers shortened by JavaScript minifiers.

Another very powerful feature available in this tab is a global search. Let's assume you've spotted a definition of some interesting function and you want to find out if it is called anywhere else in the code. Maybe it's a function contains an *eval()* call with parameter passed in url which then can be abused to execute arbitrary JavaScript code. To perform a search across all files listed in *Sources* tab, you can use CTRL+Shift+F shortcut (CMD+Option+F on macOS). In the following example, I've tried to found all references to *getAccount()* function from *AppMeasurement.js* file. As you can see, this function is called only once, in the same file, but if there is any other occurence of searched string in any of the files—you will find it on results list:



Results of getAccount() search across all files loaded by the website

Sometimes you will find that search result was found in a very, very long string (typically minified JavaScript file). You can just click on that and when DevTools will open this file, click *{}* icon and it shows you unminified version with your result—right in the correct place, even if file has thousands of lines.

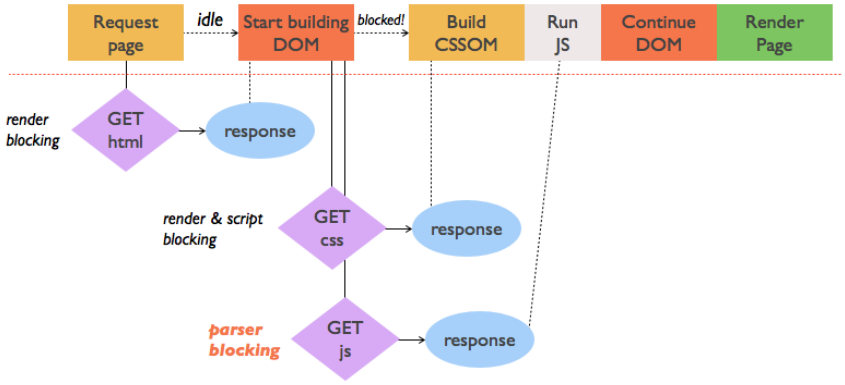
Second tab where you can investigate source code is called *Elements*. **There is one but very important difference between what can be seen in *Sources (index)* file (or what you can see with *View source* option) and content presented in *Elements* though.**

The former shows HTML file loaded from the server. *Elements* however **shows you the current DOM tree, with all elements created and added by JavaScript code.** To understand this difference, I will present another small example, but first some theory.

DOM (*Document Object Model*) is an actual representation of all of the website HTML nodes and it is a tree with a single root element

(<*html*>) with two main children elements: <*head*> and <*body*>. All other elements are just the children of either <*head*> (like <*title*> or <*meta*>) or <*body*> (<*div*>, <*p*>, <*img*> and so on).

When you open an url in the browser, HTML file is loaded first and the code is parsed by the browser engine. When browser finds <*script*> or <*style*> tag (or any other tag with *src* argument, like image or video file), it stops parsing HTML and load that file. If this is executable JavaScript, it is executed immediately. If this is stylesheet, CSS rules are applied as soon as they are parsed by CSS parser. Everything together looks like on this picture (this is very simplified, but good enough tu understand the concept):

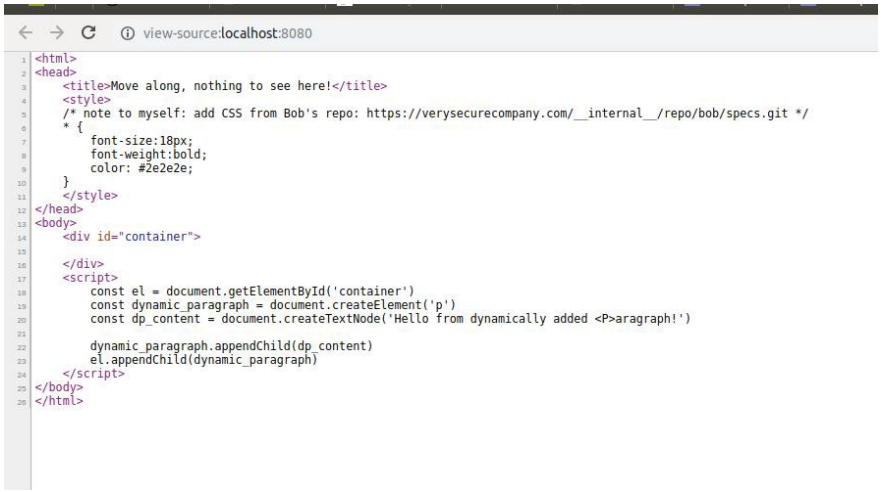


Source: https://www.sitepoint.com/optimizing-critical-rendering-path/

But what does it have to do with differences between content of *Elements* and *Sources*? Consider following example, where JavaScript adds an element to the DOM:

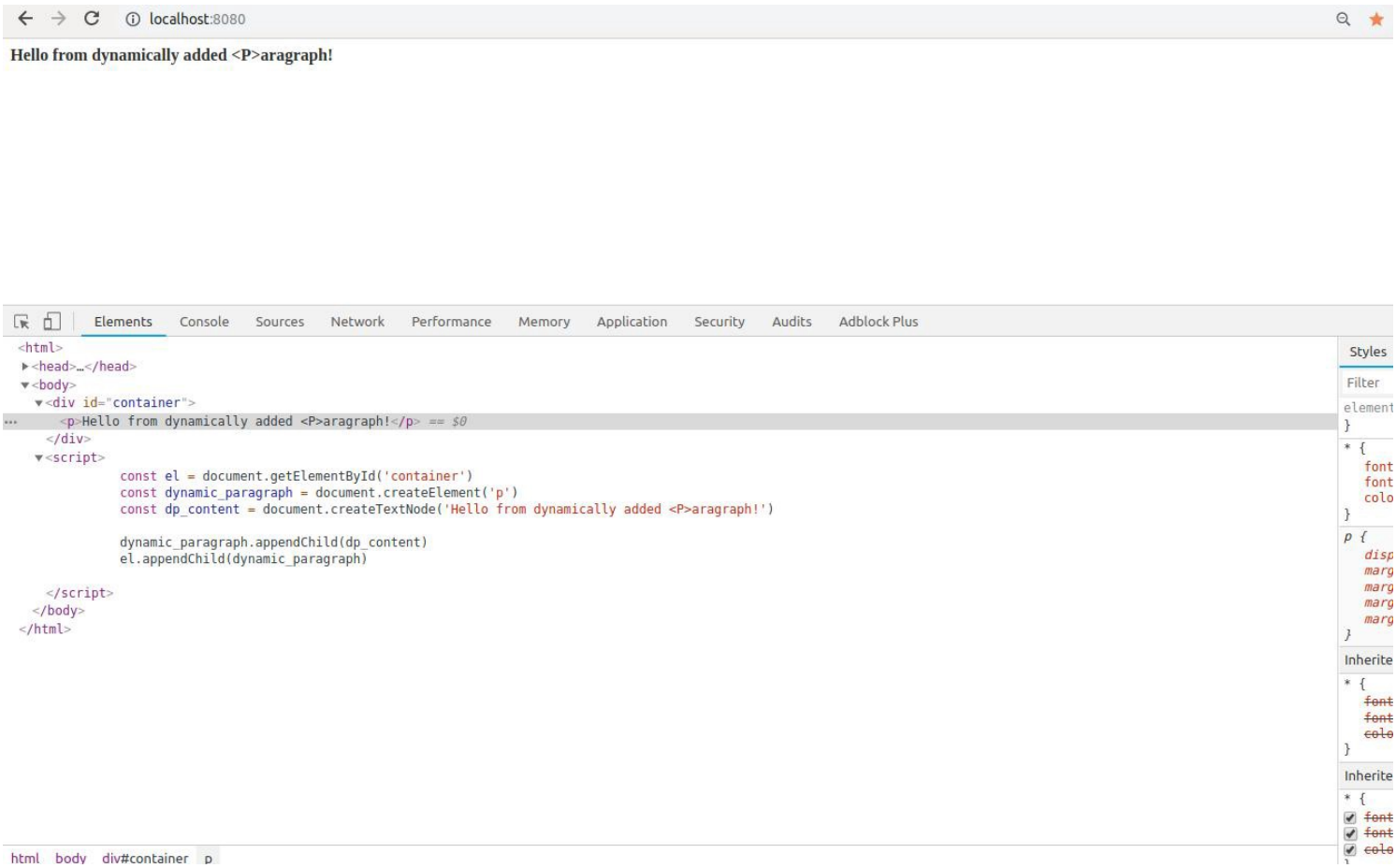
```
<html>
<head>
    <title>Dynamic P Application</title>
    <style>
    * {
        font-size:18px;
        font-weight:bold;
        color: #2e2e2e;
    }
    </style>
</head>
<body>
    <div id="container">

    </div>
    <script>
        const el = document.getElementById('container
```

When you open this page in the browser and investigate source code, you will see exactly the same content as presented in Gist above:



Source code of our simple website

This is pretty simple example of JavaScript adding new element to the DOM tree. To see the difference, use the *Elements* tab in DevTools:



When you compare what's presented in *Elements* with the version you can see by using *View source*, you can easily spot the difference: there is a <p> element visible in former, added as a child node of * * element. You can't see this element in source code earlier, because it does not exists as a part of source code.

If you deal with a Single Page Application which uses one of frameworks like AngularJS, React, Vue.js, Ember.js etc. you will see **a lot of dynamically generated content** in *Elements* tab. This content might vary, but expect forms, dynamic tables or lists with sorting and pagination, search features and plenty of elements where e.g. DOM-based XSS can be found or user input can be evaluated in template expressions (like *{{ }}* in AngularJS).

The reason is such applications often uses parameters passed via GET or POST requests or saved in cookies or browser storage to build the content of the website. Also, they create a lot of the content on their own. There is always a chance to spot a vulnerability there.

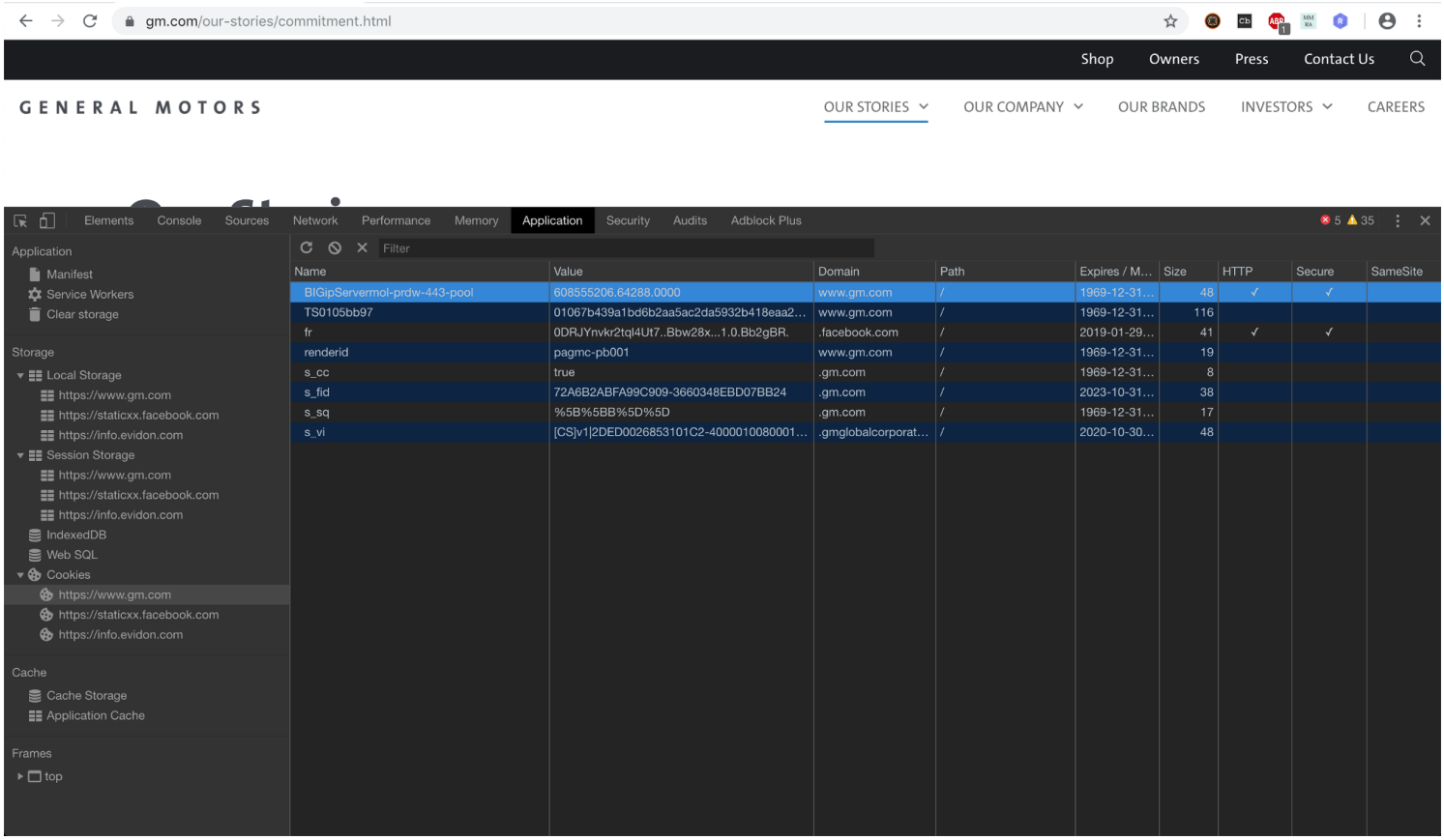
Before we will move to JavaScript itself, there is one more important thing to mention about—**always read all the HTML comments** you will find in the source as they are not rendered to the page as well. You'll be surprised how many valuable information you will be able to find.

Investigating cookies and browser Storage

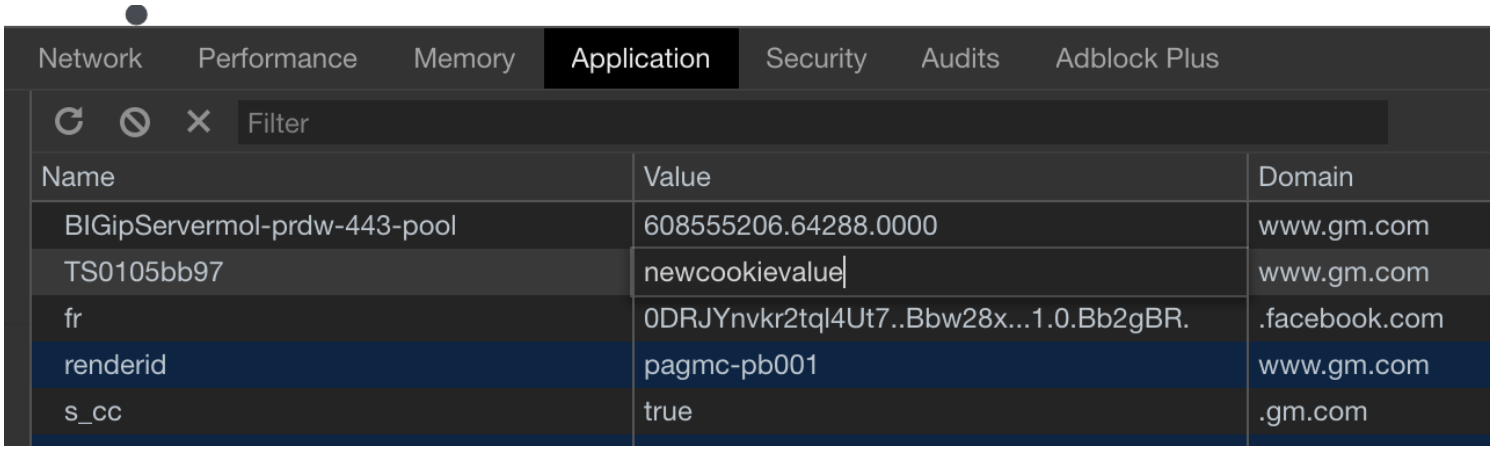
Another thing you can do with Developer Tools is to check whether website store some information on the client side. There are a couple of places that web application is able to use. The most common and popular are cookies—a small pieces of data identified by name (you can think about cookies as simple key->value map) and exchange between server and browser in HTTP requests and responses.

Browser Storage is another place where you will be able to find a lot of valuable information. There are two types of Storage: Local Storage and Session Storage. The difference between them is that content of Session Storage is lost when an application is closed (by closing the browser tab or browser itself). Local Storage has an ability to keep data as long as it is not explicitly cleaned up (data has no expiration time).

To view all information stored across different places, you can use *Application* tab in DevTools:



Using *Application* tab you can not only see the content, you have the possibility to modify, remove or add your own pairs of keys and corresponding values to see whether you are able to cause unexpected behavior or even vulnerability. It's the easiest way to verify if e.g. session token change leads to impersonate another user—just change the value of a cookie which is responisble for keeping session identifier (it's only an example, as many modern web applications use several ways to identify users and in most cases changing value of sinlge cookie is not enough to hijack another user's session):



A value of TS0105bb97 cookie modified directly in Application -> Cookies tab in DevTools

There is also one more place on this tab, where you can find JavaScript source code related to the web application: *Service Workers*. Here's one of many introduction into Service Workers you can find on web—https://developers.google.com/web/fundamentals/primers/service-workers/

This is still quite a new thing and not so many web applications use them, but they are still a valuable source of information about how web application works, especially how it works when it goes offline.

Investigating JavaScript

Now let's move to this part of the code which actually runs the whole web application (HTML and CSS are responsible only for the visual part as they can't contain any business logic. Besides some small exception like CSS expressions, which are capable to run JavaScript code and thus might be a source of XSS vulnerability if user input is used in one—there is not much you can do with plain HTML and CSS).

There are several ways to perform analysis of JavaScript code. Let's stick with the browser first. We've already revealed *Sources* tab and how *{}* feature can be used to read unminified source code. But there is more you can do with DevTools and one of the best thing is JavaScript debugger.

Using DevTools debugger

If you're not familiar with what debugging is, in general it is a possibility to stop execution of the program at particular line of the code. This allows you to see the actual variables values, what function is actually executed and how this function was called (this is available thanks to *call stack*—debugger is able to show you the exact order of function calls, like function *a()* was called by function *b()* and function *b()* was called by yet another function *c()* to give you an example). Also debugger allows you to run code step-by-step (one instruction at the time) which gives an opportunity to follow every change of the program and its state. Last but not least, debugger allows to modify program "on the fly", which means you can see what will happen when you modify variables values or even the logic of the program itself. The ability to use debugger in an effective way is one of the most important skills every programmer should have.

From bug bounty hunter perspective debugging allows you to understand better how application works and test your payloads directly at the place they can be injected. You can easily isolate vulnerable part of the program and focus on testing it with all advantages debugger gives you. As an example, imagine that you find vulnerable redirect function, but every time you try to see what's exactly going on, this function is called and browser performs the redirection to external resource and you're not able to see JavaScript from the previous page as it is now the source code of page you've been just redirected to.

Setting a breakpoint at the beginning of redirection function stops browser from performing redirection and you can now read function source code to understand how it works, figure out how you can inject your payload, if there is any encoding implemented in place and so on.

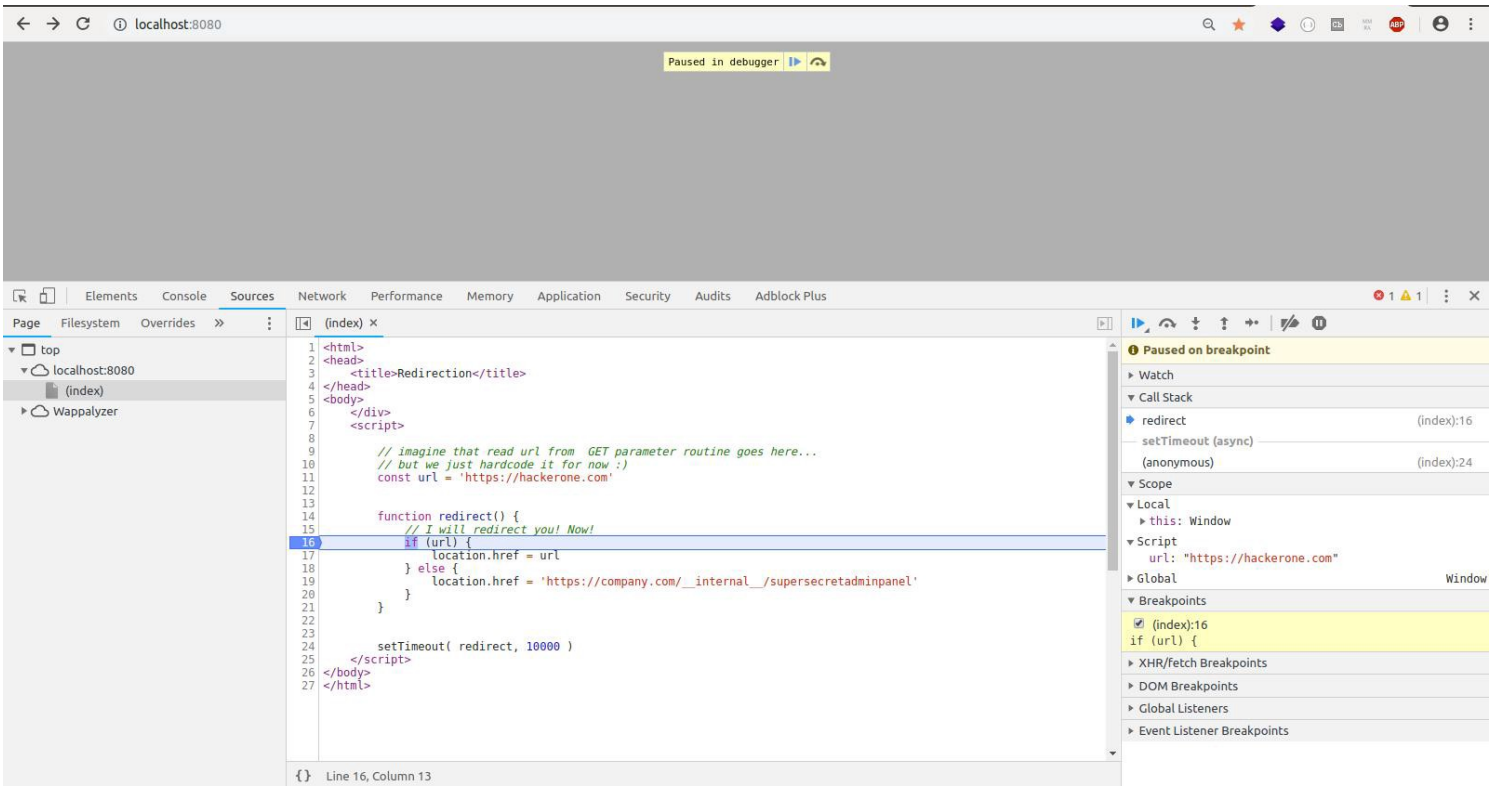
That's theory, it's time for some practice.

Here's a sample implementation of redirect feature:

```
<html>
<head>
    <title>Redirection</title>
</head>
<body>
    </div>
    <script>

        // imagine that read url from  GET parameter
        // but we just hardcode it for now :)
        const url = 'https://hackerone.com'


        function redirect() {
            // I will redirect you! Now!
            if (url) {
                location.href = url
            } else {
```

When you open this website in the browser, it redirects you to HackerOne website after 10 seconds and you won't be able to see the source of original website anymore thus there is no possibility to see what just happened.

Open Developer Tools in the browser and switch to the Sources tab, then open above example HTML file. Now you have 10 seconds to set the **breakpoint** in line 16 (*if (url) {*). To do this, just click **16** on the left bar with line numbers. When 10 seconds will pass, browser will call *redirect()* and stops the execution immediately at the line where breakpoint was set:

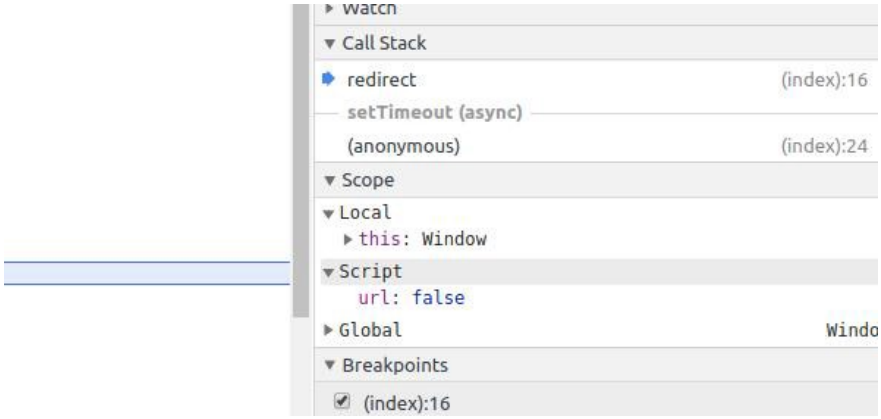


Our simple redirection application stopped in debugger

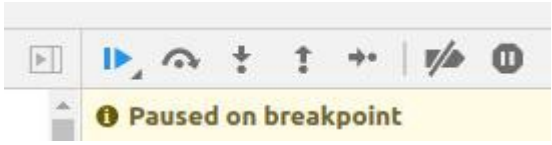
The blue line is the actual line of the code which **will be executed** when we will go forward (**it's not executed yet!** this is very important to know). On the left you can see debugger panel—it shows you where are you now (*Call stack*) and if you unfold *Script* node you will see values of all variables which are defined in the current execution scope (it's only *url* in our case).

Now you can spend as many time as you need to read and understand the code. As we already know, we will be redirected to HackerOne website, but **only if condition in line 16 will return *true***

Let's modify the value of *url* then. We will change it to something what returns *false* in JavaScript to make condition to fail (it can be empty string, 0, boolean *false* or any false expression). Let's stick with *false* (to change the value of the variable, simply click on it and put your own value):

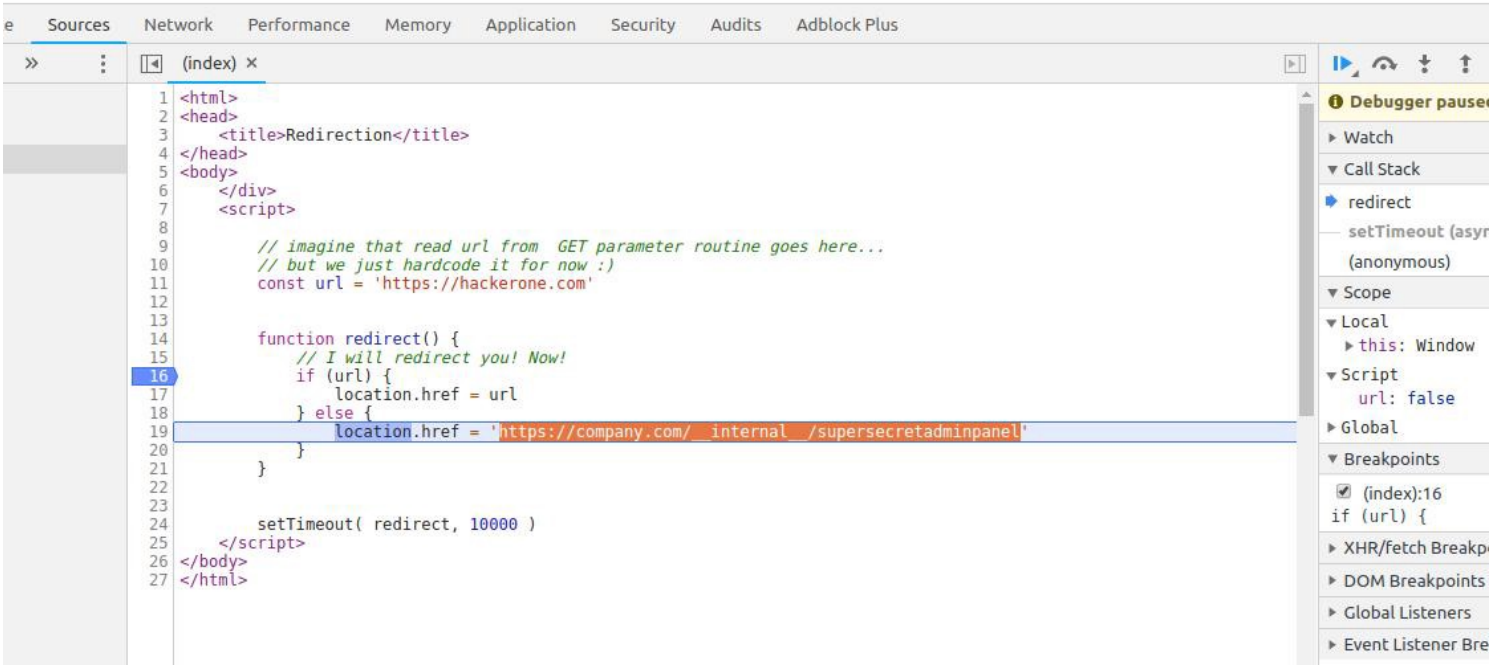


Now, let's check what has changed. We will go further by one step. Take a look at the icons at the top of the debugger panel:



The first icon will just continue to run the program, the second one allows to execute code step-by-step (we will use this one in a while). Next icons allows you to jump into the function called in the particular line (debugger does not go into function on its own as long as there is no breakpoint set there, it just executes the line with function call and move to the next line), then there is an icon which "jumps out" back from the function being executed and goes back to the point where it was called.

Now, click the second icon (make sure you've changed the value of *url* variable to *false*) and you will notice that the next line to execute is 19 (this was caused by our false condition result in line 16):



We have just changed the flow of the application

If you will press the first icon on the debugger toolbar ("Play" icon) you will notice that this time application will try to redirect you to some internal url in company.com.

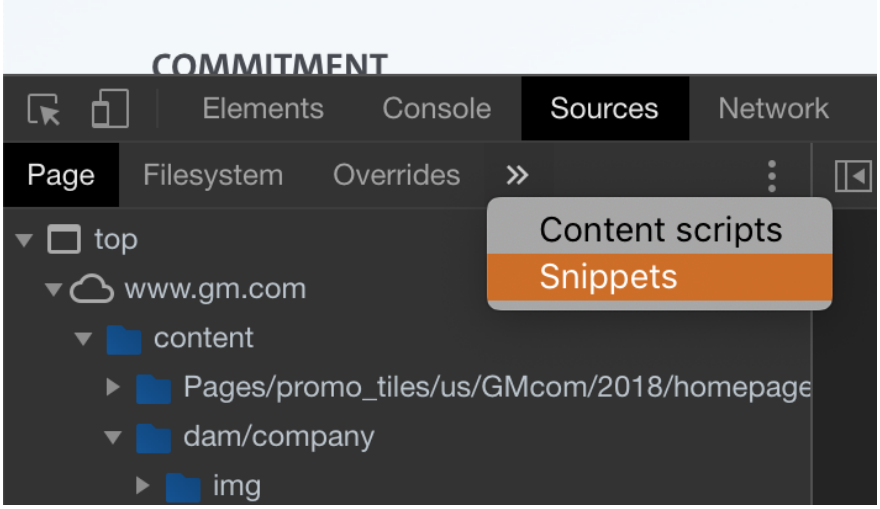
This gives you an idea that *url* parameter might be vulnerable, so you can now try to exploit this issue to find Open Redirection or Reflected XSS or dig deeper in case you suspect that value from *url* can be stored somewhere on the server side (you can find out if it's true by further investigation of the program logic).

Execute JavaScript using Snippets

Sometimes, you might want to execute only a particular fragment of the application code. It might be hard and time consuming especially when some prework has to be done to get to the point which is interesting. In that case you can use *Snippets* and run only this code you want. But remember that it is not always possible, for example when you try to run a part of the code which has some dependiences, like variables passed from other parts or when fragment of the code you want to run uses other functions defined in other files.

But let's assume you've identified a function which checks if provided value is correct and you want to focus on its logic only.

In *Sources* tab you can find a panel called *Snippets*:



Sources -> Snippets panel button

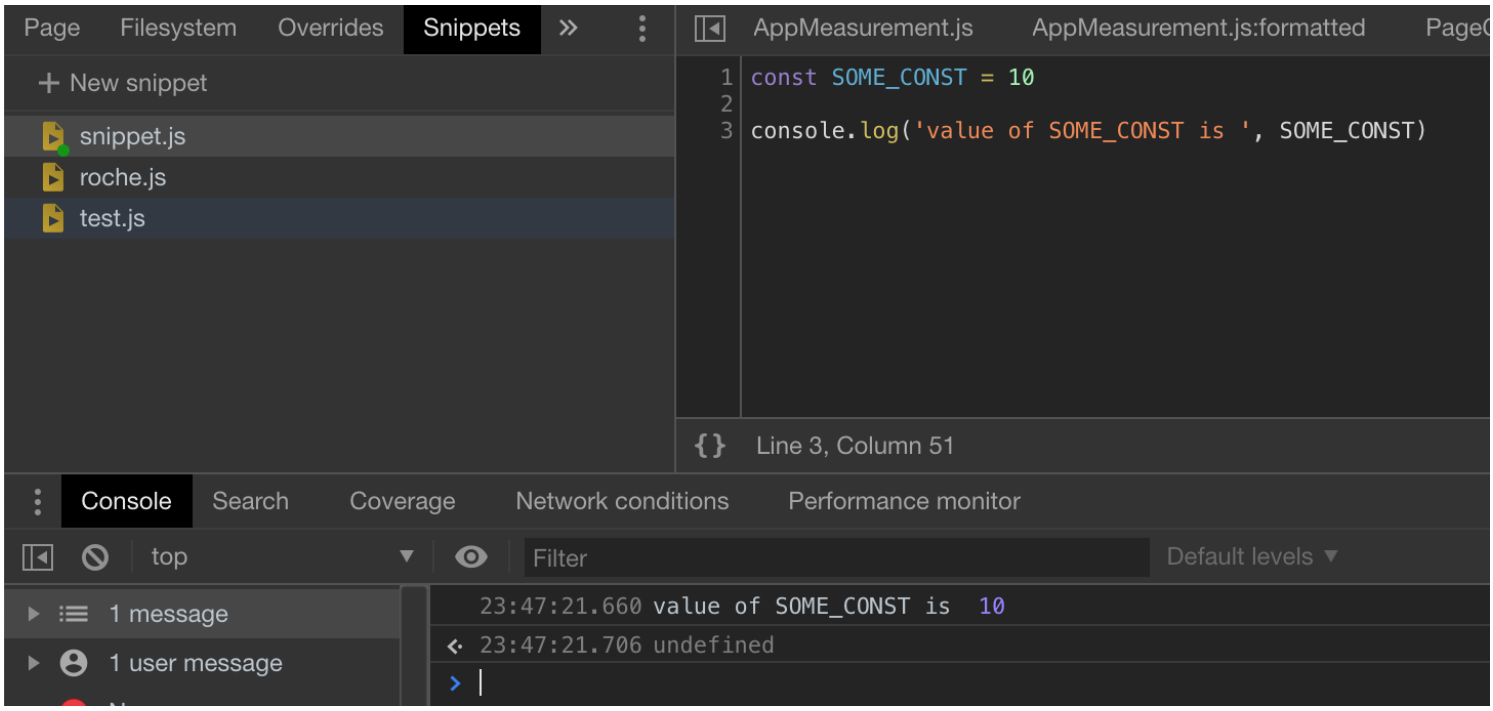
When you click on it, you will see a list of snippets (if you've already created any) and an option to create a new one. Click on this option and in the middle panel you will find a simple code editor with syntax highlighting. Every JavaScript code you put there can be run and a result is immediately printed out in the console which will appear below as soon as you run your snippet you can use "Play" icon at the bottom left of this panel or just press CMD+Enter on macOS or CTRL+Enter on other systems):

You can modify your code and run it as many time as you want, but as I mentioned at the beginnig of this post there are some cons here.

Snippets are run in the context of the page loaded in tab where you open DevTools and create and run your snippet. Also, every time you run your snippet, it is **run using the same context, which means all variables you have defined earlier and did not change their values are still there and keep the last value they have**.

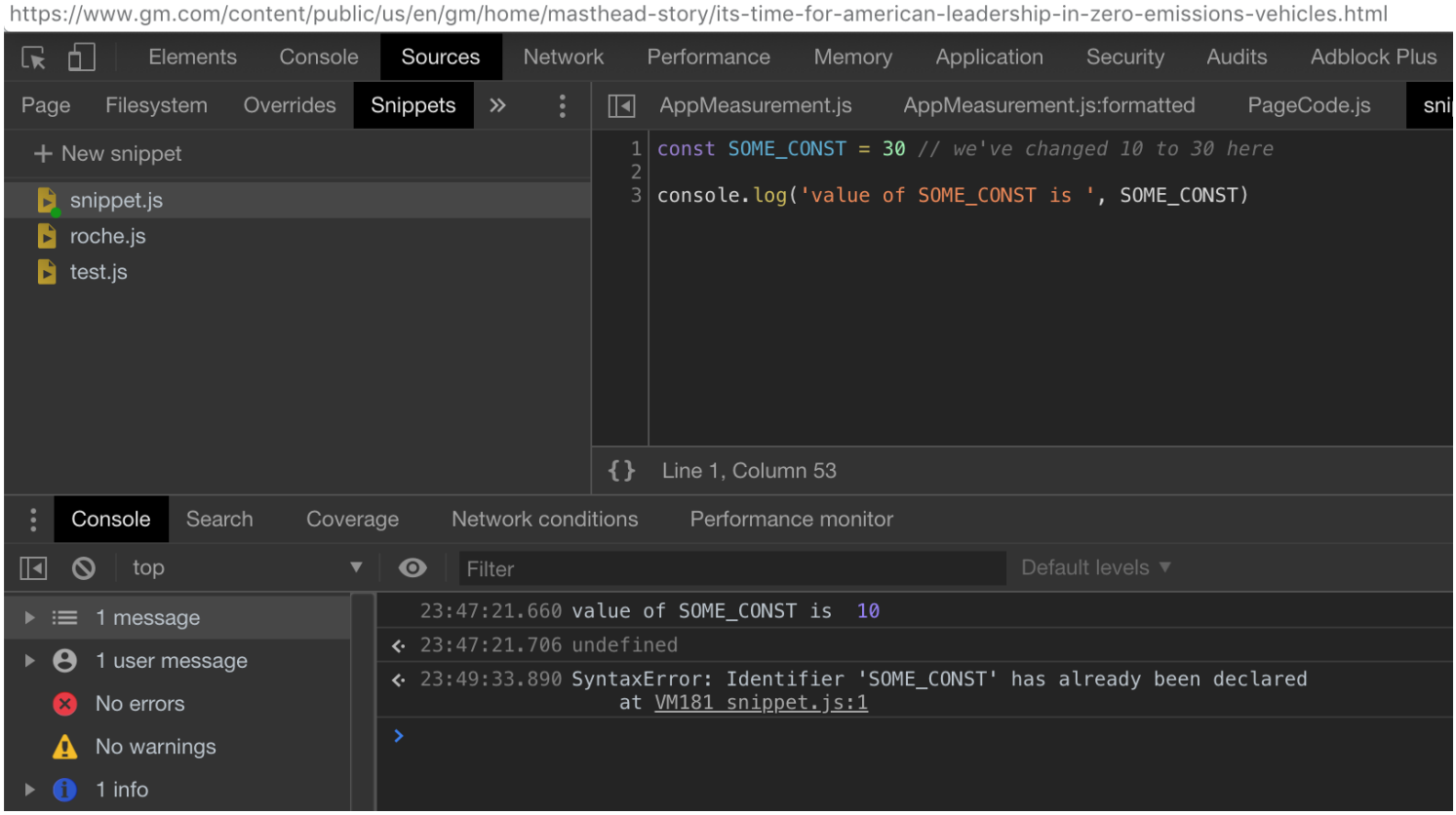
Why it is important?

Consider this example:



Snippet with constant run for the first time

In JavaScript, when you define constant using *const* keyword, you have to initialize it with the value and its value can't be changed later. As you can see, snippet works as expected, but if you will try now to change value which was used to initialize SOME_CONST and re-run the snippet, you will get a syntax error:



Syntax error caused by initialization of constant second time in the same execution context

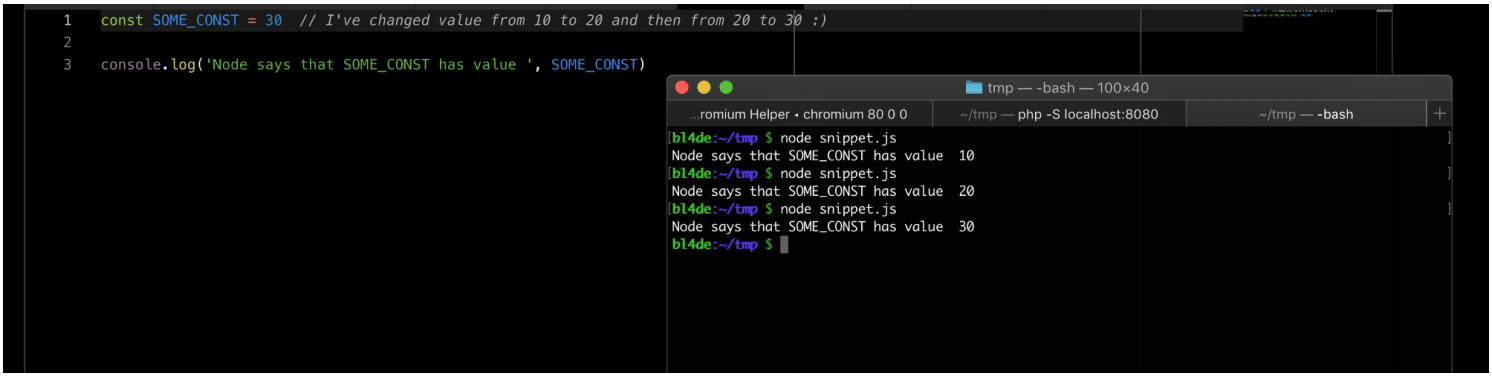
This error is caused by the fact that in this context SOME_CONST was already initialized. DevTools just "thinks" that you continue to execute code in the same execution context and it does not matter that you have edited the code.

So, if you for example pause running program with debugger (all variables, objects and functions defined by application code will now exist in the execution context) and try to create snippet in the same tab using one of existing identifiers—you will either overwrite something from original web application code or get an error if you will use identifier of something which can't be reinitialized (like constant). To be able to re-run the snippet, you need first to reload the page in the browser to provide a fresh, empty execution context (browser does not remember the state of the web application between reloads as reloading causes the whole process of loading resources, building DOM tree etc.).

To avoid such problems, instead of using *Snippets* (which are still pretty handy tool as you can simply open new tab with DevTools and

create your snippet there) you can run your code using NodeJS JavaScript runtime environment.

To do this, just put the code you want to run in a new JavaScript file and run it with NodeJS (make sure you have installed it first) using terminal:



The same code run three times, every time with the new value used as SOME_CONST initializer

I've run this code three times, every time changing the value of *SOME_CONST*. As you can see, there are no errors and every execution was successful and prints the correct result.

This behavior is caused by the nature of NodeJS—every time you run JavaScript code using it, it creates new execution context, so there is no possibility to run the same code twice using the same execution context.

Sources and execution sinks

When you look at JavaScript code, there are two main things you should focus on in the first place.

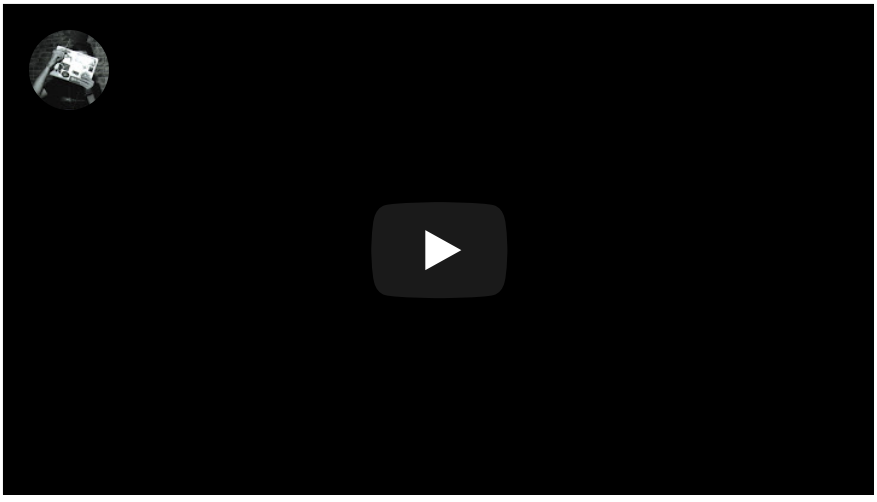
First one are **sources,** and this term describes every point where user provided input meets application code. That can be an argument passed via GET in url, cookie read by application or content of Local Storage used in business logic.

Second one is called an **execution sink.** This term means all JavaScript syntax elements or HTML APIs features which lead to **execute code passed to them as an argument.** The one obvious

example here is JavaScript function *eval(code_to_evaluate)* which evaluates (executes) code passed to it as an argument. Another example is *setTimeout(function_to_execute, timeout_in_miliseconds)* which executes function passed as a first argument after the time passed as the second argument.

The process of discovering a vulnerability in web application is to find a connection between source and execution sink which actually process this source. In the example where I've presented how to use a debugger, an *url* passed as an argument (**source**) was used directly in *location.href* (**execution sink**). Another example here might be a function which evaluates data put by user in HTML form input field (JavaScript can read this using DOM API, e.g. *document.getElementById('input_id').value* and assign this to the variable—that will be the source in this case) and then pass this value into another element *innerHTML()* method updating actual DOM in the browser window (and that will be an execution sink).

There is an awesome video about this topic made by @LiveOverflow available on his YouTube channel. I recommend to stop reading this post right now and go to watch this video to become familiar with this concept (it's ~8 minutes length)



https://www.youtube.com/watch?v=ZaOtY4i5w_U

There are many sources and execution sinks existing in web application depends on its business logic (think of form fields, url parameters, cookies, browser storage, WebSockets etc.). But the most important thing is that are they actually being used in any of the

execution sinks. There are plenty of them, including *location* properties like *href* or *hash*, *window.open()*, *document.write()*, or DOM methods like *innerHTML* or *appendChild*. They all can be used to execute arbitrary code, make redirects or perform other types of injections.

For easy identification of such code patterns I've created a tool, **nodestructor** It just inspects JavaScript files (both single file or all JavaScript files in the directory provided as an argument) and finds patterns that are well known as execution sinks (or sources). **Do not expect that every single line of code identified by nodestructor can be immediately or easily exploited**—all depends on what's between source and execution sink (sanitization, encoding, parsing, transforming data into object, string manupulation etc.). The main purpose of this tool is to provide an easy to use and quick way to find such patterns in large codebases.

Let me show you a quick usage example. First, we need a JavaScript file to inspect. The file I am going to check is *AppMeasurement.js* from GM.com website we've seen earlier. I simply copied this file from browser (after unminifying it first) and paste into code editor and save locally in */tmp* folder.

In the terminal, I run *nodestructor* on *AppMeasurement.js* file (with *-H* option to include search for various HTML5 APIs patterns as well):

```
[bl4de:~/tmp $ nodestructor -H AppMeasurement.js

                    (               )                   )
                   )\ )   (      ( /( (      (        ( /(      (
          (      (   (()/(  ))\ (   )\()))(    ))\   (   )\()) (   )(
          )\ )   )\   ((_))/((_))\ (_))/((()\  /((_)  )\ (_))/  )\ (()\
          _(_/(  ((_)  _| |(_)) ((_)| |_  ((_)(_))(  ((_)| |_  ((_) ((_)
          | ' \))/ _ \/ _` |/ -_)(_-<|  _|| '_|| || |/ _| |  _|/ _ \| '_|
          |_||_| \___/\__,_|\___|/__/ \__||_|   \_,_|\__|  \__|\___/|_|

#####    static code analysis for Node.js and other JavaScript apps        #####
#####    GitHub.com/bl4de | twitter.com/_bl4de | hackerone.com/bl4de       #####

example usages:
            $ ./nodestructor filename.js
            $ ./nodestructor -R ./dirname
            $ ./nodestructor -R ./dirname --skip-node-modules --skip-test-files
            $ ./nodestructor -R ./node_modules --exclude=babel,lodash,ansi
            $ ./nodestructor -R ./node_modules --include=body-parser,chalk,commander
            $ ./nodestructor -R ./node_modules --pattern="obj.dangerousFn\("


FILE: AppMeasurement.js

::  line 8 :: .location.host  code pattern identified:
s.linkInternalFilters = "javascript:," + location.hostname;
...
::  line 17 :: .location.host  code pattern identified:
    var domain = location.hostname.toLowerCase();
...
::  line 143 :: .innerText  code pattern identified:
          (f = g(k(a.innerText || a.textContent), e.linkExclusions)) || (s(a, c = [], b = {
...
::  line 143 :: .textContent  code pattern identified:
          (f = g(k(a.innerText || a.textContent), e.linkExclusions)) || (s(a, c = [], b = {
...
::  line 187 :: .location.host  code pattern identified:
        s = n.location; m && m.location && s && "" + m.location != "" + s && n.location && "" + m.location != "" + n.loc...
::  line 227 :: .location.host  code pattern identified:
        var c = k.location.hostname, b = a.fpCookieDomainPeriods, d;
...
::  line 641 :: .innerText  code pattern identified:
        g = 2) : "INPUT" == b || "SUBMIT" == b ? (c.value ? e = c.value : c.innerText ? e = c.innerText : c.textContent ...
::  line 641 :: .textContent  code pattern identified:
        g = 2) : "INPUT" == b || "SUBMIT" == b ? (c.value ? e = c.value : c.innerText ? e = c.innerText : c.textContent ...
::  line 694 :: .location.host  code pattern identified:
          a.Na(l) && (a.linkInternalFilters || (a.linkInternalFilters = k.location.hostname),
...
::  line 1119 :: .document.referrer  code pattern identified:
        a.referrer = f || void 0 === f ? void 0 === f ? "" : f : n.document.referrer),
...
::  line 1285 :: .localStorage.  code pattern identified:
              (b = k.localStorage.getItem(a.qa())) && (c = k.JSON.parse(b))
...
::  line 1444 :: .appendChild(  code pattern identified:
          f.firstChild ? f.insertBefore(b, f.firstChild) : f.appendChild(b);
...
::  line 1461 :: .localStorage.  code pattern identified:
              k.localStorage.removeItem(a.qa()),
...
::  line 1470 :: .localStorage.  code pattern identified:
              k.localStorage.setItem(a.qa(), k.JSON.stringify(c)),
...
::  line 1570 :: .innerHTML  code pattern identified:
              b.innerHTML = "\x3c!--[if IE " + a + "]><span></span><![endif]--\x3e";
...
::  line 1607 :: .navigator.userAgent  code pattern identified:
          var v = navigator.userAgent;
...
::  line 1682 :: .navigator.userAgent  code pattern identified:
        a.b && a.b.attachEvent ? a.b.attachEvent("onclick", a.v) : a.b && a.b.addEventListener && (navigator && (0 <= na...
Identified 17 code pattern(s)

-------------------------------------------------------------------------------------------

 1 file(s) scanned in total

Identified 17 code pattern(s) in 1 file(s)
```

Result of nodestructor run on AppMeasurement.js file

As you might notice, the tool identified a couple of potential execution sinks. Most of them are false positives, but for just a presentation purposes let's try to focus on second item, which looks like *domain* variable initialization with *location.hostname.toLowerCase()* function result.

It would be nice to have a possibility to follow all occurences of this variable across the file to find out if it's used in any of execution sink later. You can use built-in Visual Studio Code features like *Find all references* or just to search for the string *domain*.

I wanted to go a little bit further though so some time ago I've started to work on my own tool to perform such task—a simple static code analysis for JavaScript files. It has no fancy name yet :) and it is still in very early stage of development (it's more PoC right now than actual working tool to be honest…), but just to give you a quick overview I will run this tool against *domain* variable (that's the only available option so far, but as I've mentioned it is just a very beggining of my work on this tool, so e.g. a filename to look into is hardcoded :P)

Result of simple analysis to find out all usages of domain variable

As you can see, tool identifies where variable is defined and when and how it is used. I hope to make this tool capable to perform much more sofisticated analysis, like looking for variable usage in different

scopes (e.g. if it's passed as an argument to any function or if it is actually used as an argument for any execution sink).

Summary

Web browser is a very powerful tool. Sometimes, it is **the only tool you will need** to read the source code and fully understand how application works, identify its weaknesses, make some tests to find vulnerabilities or just to see how it works and learn something new.

I hope my post will help you to find out how to use a very powerful feature which is Developer Tools. You can find a lot of great resources about how to use its full power at this url https://developers.google.com/web/tools/chrome-devtools/

If you have any questions or just want to say how my post sucks :D— do not hesitate to reach me on Twitter

Thank you for reading, I wish you a lot of awesome bugs to be found :)

Happy Hacking!

bl4de, 1st of November, 2018