JavaScript Deobfuscation1

Introduction

Welcome to the JavaScript Deobfuscation module!

Code deobfuscation is an important skill to learn if we want to be skilled in code analysis and reverse engineering. During red/blue team exercises, we often come across obfuscated code that wants to hide certain functionalities, like malware that utilizes obfuscated JavaScript code to retrieve its main payload. Without understanding what this code is doing, we may not know what exactly the code is doing, and hence may not be able to complete the red/blue team exercise.

In this module, we start by learning the general structure of an HTML page and then will locate JavaScript code within it. Once we do that, we will learn what obfuscation is, how it is done, and where it is used and follow that by learning how to deobfuscate such code. Once the code is deobfuscated, we will attempt to understand its general usage to replicate its functionality and uncover what it does manually.

The following topics will be discussed:

- Locating JavaScript code
- Intro to Code Obfuscation
- How to Deobfuscate JavaScript code
- How to decode encoded messages
- Basic Code Analysis
- Sending basic HTTP requests

Source Code

Most websites nowadays utilize JavaScript to perform their functions. While `HTML` is used to determine the website's main fields and parameters, and `CSS` is used to determine its design, `JavaScript` is used to perform any functions necessary to run the website. This happens in the background, and we only see the pretty front-end of the website and interact with it.

Even though all of this source code is available at the client-side, it is rendered by our browsers, so we do not often pay attention to the HTML source code. However, if we wanted to understand a certain page's client-side functionalities, we usually start by taking a look at the page's source code. This section will show how we can uncover the source code that contains all of this and understand its general usage.

HTML

We will start by starting the exercise below, open Firefox in our PwnBox, and visit the url shown in the question:

As we can see, the website says `Secret Serial Generator`, without having any input fields or showing any clear functionality. So, our next step is to peak at its source code. We can do that by pressing `[CTRL + U]`, which should open the source view of the website:

```
</html>
<!DOCTYPE html>

<head>
    <title>Secret Serial Generator</title>
    <style>
        *,
        html {
            margin: 0;
            padding: 0;
            border: 0;
        }

        html {
            width: 100%;
            height: 100%;
        }

```

As we can see, we can view the `HTML` source code of the website.

CSS

`CSS` code is either defined `internally` within the same `HTML` file between `<style>` elements, or defined `externally` in a separate `.css` file and referenced within the `HTML` code.

In this case, we see that the `CSS` is internally defined, as seen in the code snippet below:

```
    <style>
        *,
        html {
            margin: 0;
            padding: 0;
            border: 0;
        }
        ...SNIP...
        h1 {
            font-size: 144px;
        }
        p {
            font-size: 64px;
        }
    </style>
```

If a page `CSS` style is externally defined, the external `.css` file is referred to with the `<link>` tag within the HTML head, as follows:

```
<head>
    <link rel="stylesheet" href="style.css">
</head>
```

JavaScript

The same concept applies to `JavaScript`. It can be internally written between `<script>` elements or written into a separate `.js` file and referenced within the `HTML` code.

We can see in our `HTML` source that the `.js` file is referenced externally:

```
<script src="secret.js"></script>
```

We can check out the script by clicking on `secret.js`, which should take us directly into the script. When we visit it, we see that the code is very complicated and cannot be comprehended:

```
eval(function (p, a, c, k, e, d) { e = function (c) { '...SNIP... |true|function'.split('|'), 0, {}))
```

The reason behind this is `code obfuscation`. What is it? How is it done? Where is it used?

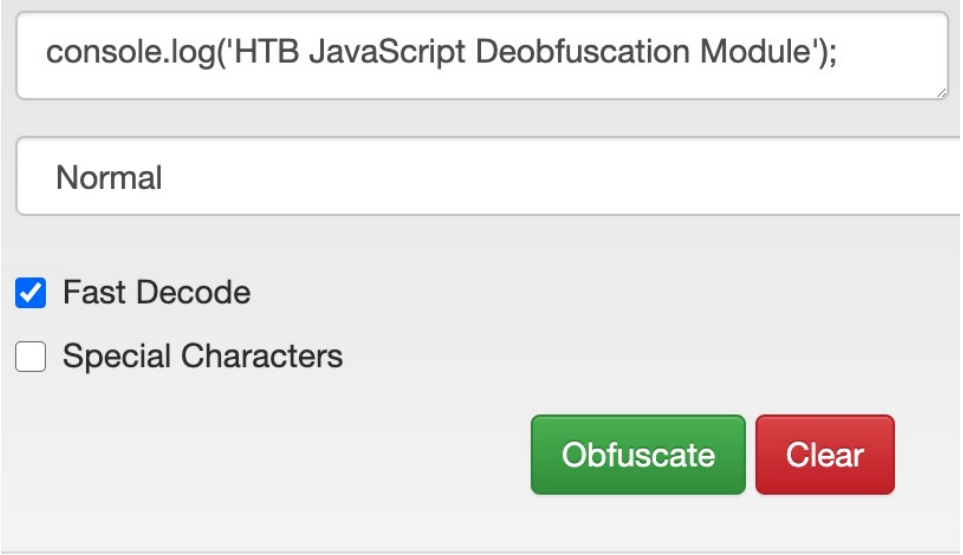
Code Obfuscation

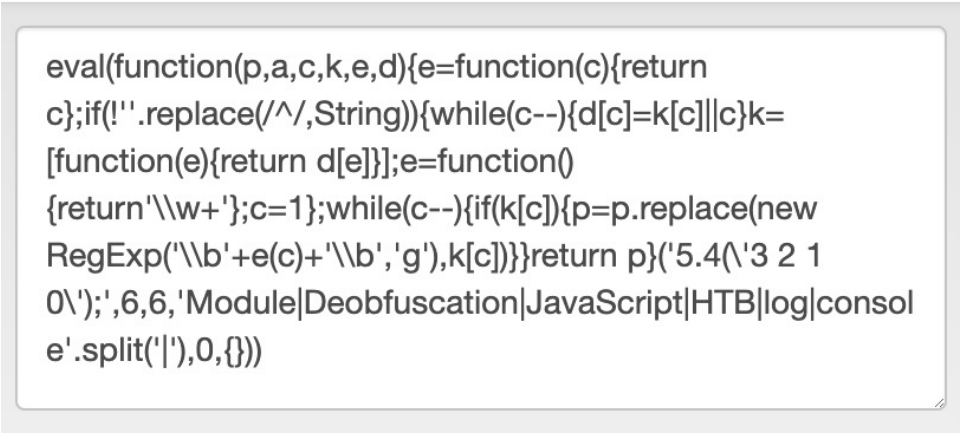
Before we start learning about `deobfuscation`, we must first learn about `code obfuscation`. Without understanding how code is obfuscated, we may not be able to successfully deobfuscate the code, especially if it was obfuscated using a custom obfuscator.

What is obfuscation

Obfuscation is a technique used to make a script more difficult to read by humans but allows it to function the same from a technical point of view, though performance may be slower. This is usually achieved automatically by using an obfuscation tool, which takes code as an input, and attempts to re-write the code in a way that is much more difficult to read, depending on its design.

For example, code obfuscators often turn the code into a dictionary of all of the words and symbols used within the code and then attempt to rebuild the original code during execution by referring to each word and symbol from the dictionary. The following is an example of a simple JavaScript code being obfuscated:





Codes written in many languages are published and executed without being compiled in `interpreted` languages, such as `Python`, `PHP`, and `JavaScript`. While `Python` and `PHP` usually reside on the server-side and hence are hidden from end-users, `JavaScript` is usually used within browsers at the `client-side`, and the code is sent to the user and executed in cleartext. This is why obfuscation is very often used with `JavaScript`.

Use Cases

There are many reasons why developers may consider obfuscating their code. One common reason is to hide the original code and its functions to prevent it from being reused or copied without the developer's permission, making it more difficult to reverse engineer the code's original functionality. Another reason is to provide a security layer when dealing with authentication or encryption to prevent attacks on vulnerabilities that may be found within the code.

`It must be noted that doing authentication or encryption on the client-side is not recommended, as code is more prone to attacks this way.`

The most common usage of obfuscation, however, is for malicious actions. It is common for attackers and malicious actors to obfuscate their malicious scripts to prevent Intrusion Detection and Prevention systems from detecting their scripts. In the next section, we will learn how to obfuscate a simple JavaScript code and attempt running it before and after obfuscation to note any differences.

Basic Obfuscation

Code obfuscation is usually not done manually, as there are many tools for various languages that do automated code obfuscation. Many online tools can be found to do so, though many malicious actors and professional developers develop their own obfuscation tools to make it more difficult to deobfuscate.

Running JavaScript code

Let us take the following line of code as an example and attempt to obfuscate it:

```
console.log('HTB JavaScript Deobfuscation Module');
```

First, let us test running this code in cleartext, to see it work in action. We can go to JSConsole, paste the code and hit enter, and see its output:

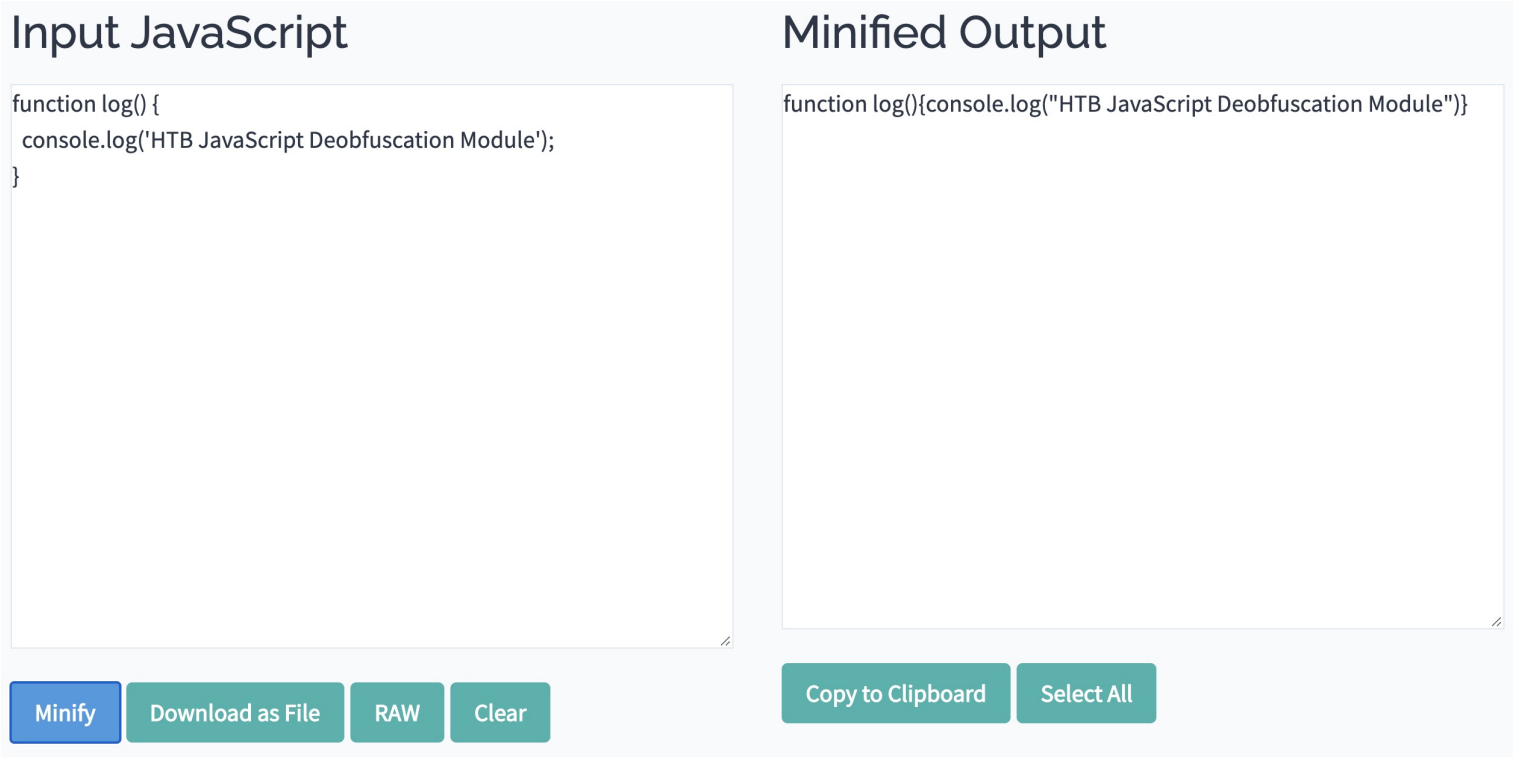
```
Use :help to show jsconsole commands
version: 2.1.2

> console.log('HTB JavaScript Deobfuscation Module');

HTB JavaScript Deobfuscation Module

<· undefined
```

We see that this line of code prints `HTB JavaScript Deobfuscation Module`, which is done using the `console.log()` function.

Minifying JavaScript code

A common way of reducing the readability of a snippet of JavaScript code while keeping it fully functional is JavaScript minification. `Code minification` means having the entire code in a single (often very long) line. `Code minification` is more useful for longer code, as if our code only consisted of a single line, it would not look much different when minified.

Many tools can help us minify JavaScript code, like javascript-minifier. We simply copy our code, and click `Minify`, and we get the minified output on the right:

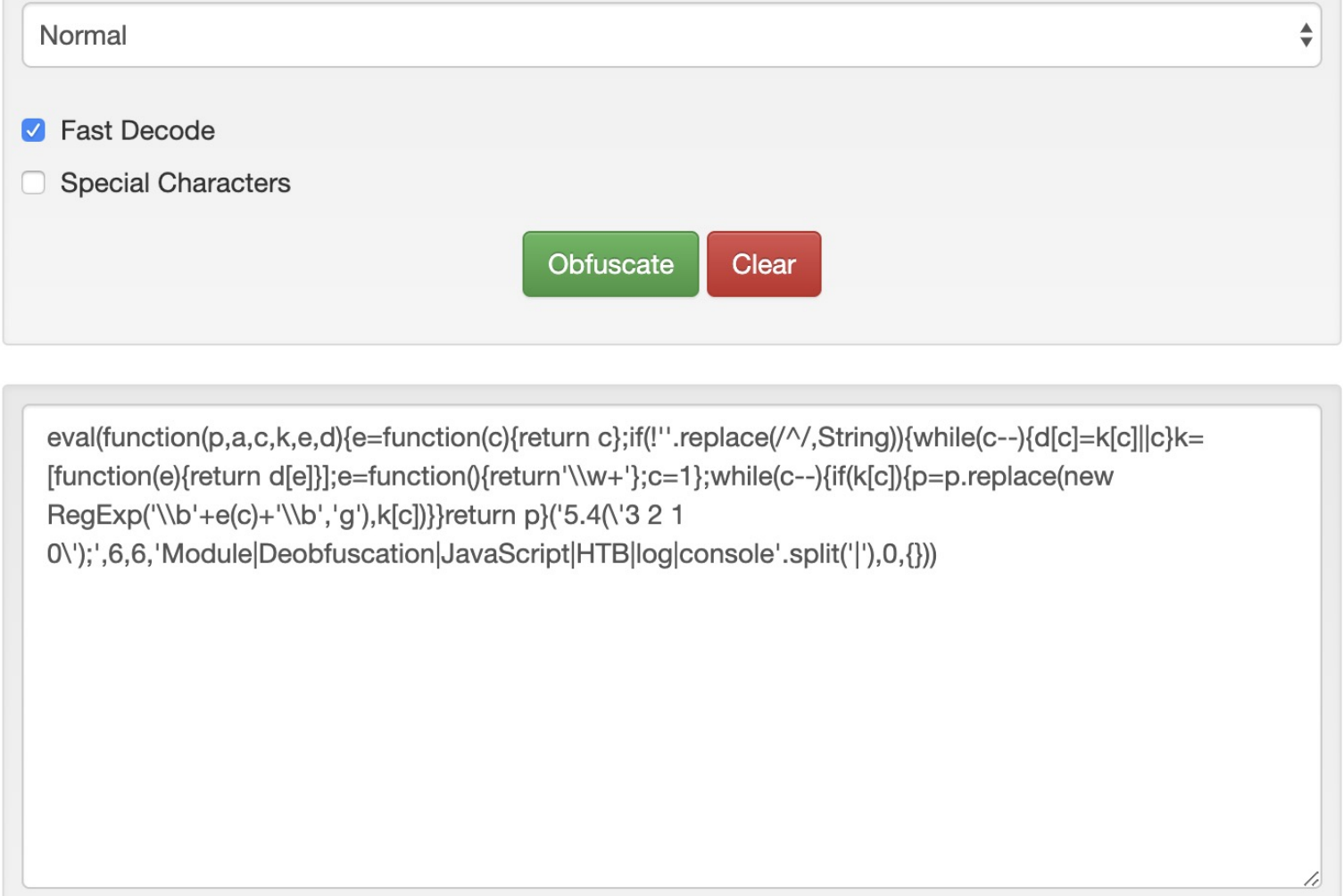


Once again, we can copy the minified code to JSConsole, and run it, and we see that it runs as expected. Usually, minified JavaScript code is saved with the extension `.min.js`.

Note: Code minification is not exclusive to JavaScript, and can be applied to many other languages, as can be seen on javascript-minifier.

Packing JavaScript code

Now, let us obfuscate our line of code to make it more obscure and difficult to read. First, we will try BeautifyTools to obfuscate our code:




```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return d[e]}];e=function()
{return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('5.4(\'3 2 1
0\');',6,6,'Module|Deobfuscation|JavaScript|HTB|log|console'.split('|'),0,{}))
```

We see that our code became much more obfuscated and difficult to read. We can copy this code into https://jsconsole.com, to verify that it still does its main function:

```
Use :help to show jsconsole commands
version: 2.1.2
> eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('5.4(\'3 2 1
0\');',6,6,'Module|Deobfuscation|JavaScript|HTB|log|console'.split('|'),0,{}))
HTB JavaScript Deobfuscation Module
```

We see that we get the same output.

Note: The above type of obfuscation is known as "packing", which is usually recognizable from the six function arguments used in the initial function "function(p,a,c,k,e,d)".

A `packer` obfuscation tool usually attempts to convert all words and symbols of the code into a list or a dictionary and then refer to them using the `(p,a,c,k,e,d)` function to re-build the original code during execution. The `(p,a,c,k,e,d)` can be different from one packer to another. However, it usually contains a certain order in which the words and symbols of the original code were packed to know how to order them during execution.

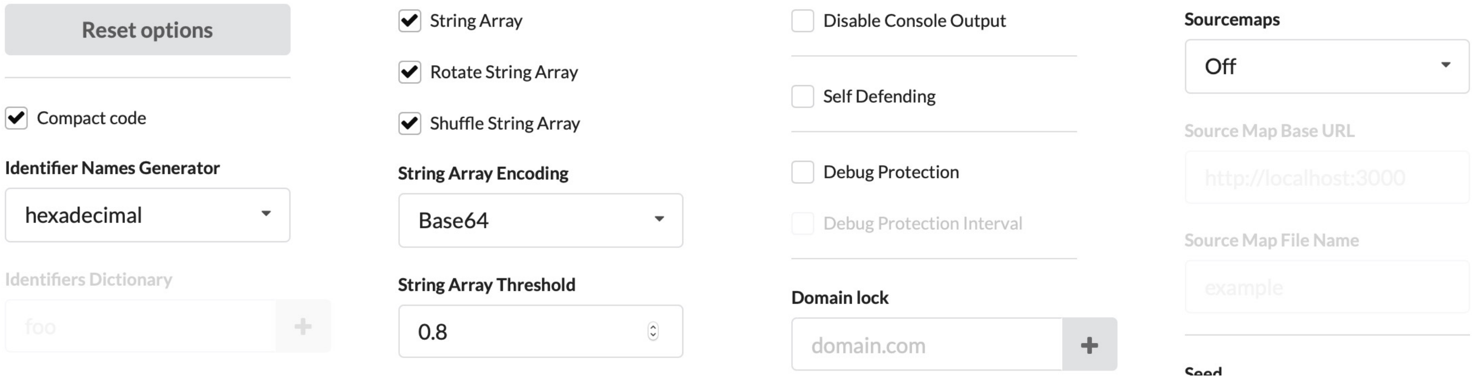
While a packer does a great job reducing the code's readability, we can still see its main strings written in cleartext, which may reveal some of its functionality. This is why we may want to look for better ways to obfuscate our code.

Advanced Obfuscation

So far, we have been able to make our code obfuscated and more difficult to read. However, the code still contains strings in cleartext, which may reveal its original functionality. In this section, we will try a couple of tools that should completely obfuscate the code and hide any remanence of its original functionality.

Obfuscator

Let's visit https://obfuscator.io. Before we click `obfuscate`, we will change `String Array Encoding` to `Base64`, as seen below:



Now, we can paste our code and click `obfuscate`:

We get the following code:

```
var _0x1ec6=['Bg9N','sfrciePHDMfty3jPChqGrgvVyMz1C2nHDgLVBIbnB2r1Bgu='];(function(_0x13249d,_0x1ec6e5){var _0x14f83b=function(_0x3f720f){while(--_0x3f720f)
{_0x13249d['push'](_0x13249d['shift']());}};_0x14f83b(++_0x1ec6e5);}(_0x1ec6,0xb4));var _0x14f8=function(_0x13249d,_0x1ec6e5){_0x13249d=_0x13249d-0x0;var
_0x14f83b=_0x1ec6[_0x13249d];if(_0x14f8['eOTqeL']===undefined){var _0x3f720f=function(_0x32fbfd){var
_0x523045='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/=',_0x4f8a49=String(_0x32fbfd)['replace'](/=+$/,'');var _0x1171d4='';for(var
_0x44920a=0x0,_0x2a30c5,_0x443b2f,_0xcdf142=0x0;_0x443b2f=_0x4f8a49['charAt'](_0xcdf142++);~_0x443b2f&&(_0x2a30c5=_0x44920a%0x4?
_0x2a30c5*0x40+_0x443b2f:_0x443b2f,_0x44920a++%0x4)?_0x1171d4+=String['fromCharCode'](0xff&_0x2a30c5>>(-0x2*_0x44920a&0x6)):0x0)
{_0x443b2f=_0x523045['indexOf'](_0x443b2f);}return _0x1171d4;};_0x14f8['oZlYBE']=function(_0x8f2071){var _0x49af5e=_0x3f720f(_0x8f2071);var _0x52e65f=
[];for(var _0x1ed1cf=0x0,_0x79942e=_0x49af5e['length'];_0x1ed1cf<_0x79942e;_0x1ed1cf++){_0x52e65f+='%'+('00'+_0x49af5e['charCodeAt'](_0x1ed1cf)['toString']
(0x10))['slice'](-0x2);}return decodeURIComponent(_0x52e65f);},_0x14f8['qHtbNC']={},_0x14f8['eOTqeL']=!![];}var _0x20247c=_0x14f8['qHtbNC'][_0x13249d];return
_0x20247c===undefined?(_0x14f83b=_0x14f8['oZlYBE'](_0x14f83b),_0x14f8['qHtbNC'][_0x13249d]=_0x14f83b):_0x14f83b=_0x20247c,_0x14f83b;};console[_0x14f8('0x0')]
(_0x14f8('0x1'));
```

This code is obviously more obfuscated, and we can't see any remnants of our original code. We can now try running it in https://jsconsole.com to verify that it still performs its original function. Try playing with the obfuscation settings in https://obfuscator.io to generate even more obfuscated code, and then try rerunning it in https://jsconsole.com to verify it still performs its original function.

More Obfuscation

Now we should have a clear idea of how code obfuscation works. There are still many variations of code obfuscation tools, each of which obfuscates the code differently. Take the following JavaScript code, for example:

```
[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]][([][(![]+[])[+[]]+([![]]+[][[]])
[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+
[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+
[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(!
...SNIP...
[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+([][[]]+
[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+
[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+
[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]]](!+[]+!+[]+[+[]])))()
```

We can still run this code, and it would still perform its original function:

```
[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+
[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]](!+[]+!+[]+!+[]+[+!+[]])[+!+[]]+
(!![]+[])[!+[]+!+[]+!+[]])()(([]+[])[([![]]+[][[]])[+!+[]+[+[]]]+(!![]+[])[+[]]+(![]+[])[+!+[]]+(![]+[])[!+[]+!+[]]+([![]]+[][[]])[+!+[]+[+[]]]+([][(![]+[])[+[]]+([!
[]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(![]+[])[!+[]+!+[]+!+[]]]()[+[]])[+[]]+[!+
[]+!+[]+!+[]]+(+(+!+[]+[+!+[]]))[(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+
[])[+!+[]]])[+!+[]+[+[]]]+(+![]+([]+[])[([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]+
[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+([]
[[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+
[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+
[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([![]]+[][[]])[+!+[]+[+[]]]+([][[]]+[])
[+!+[]]+(+![]+[![]]+([]+[])[([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]+[])[!+[]+!+
[]+!+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+([][[]]+[])
[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!!
[]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!!
[]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]](!+[]+!+[]+[+[]])))()
```

```
HTB JavaScript Deobfuscation Module
```

```
undefined
```

Note: The above code was snipped as the full code is too long, but the full code should successfully run.

We can try obfuscating code using the same tool in JSF, and then rerunning it. We will notice that the code may take some time to run, which shows how code obfuscation could affect the performance, as previously mentioned.

There are many other JavaScript obfuscators, like JJ Encode or AA Encode. However, such obfuscators usually make code execution/compilation very slow, so it is not recommended to be used unless for an obvious reason, like bypassing web filters or restrictions.

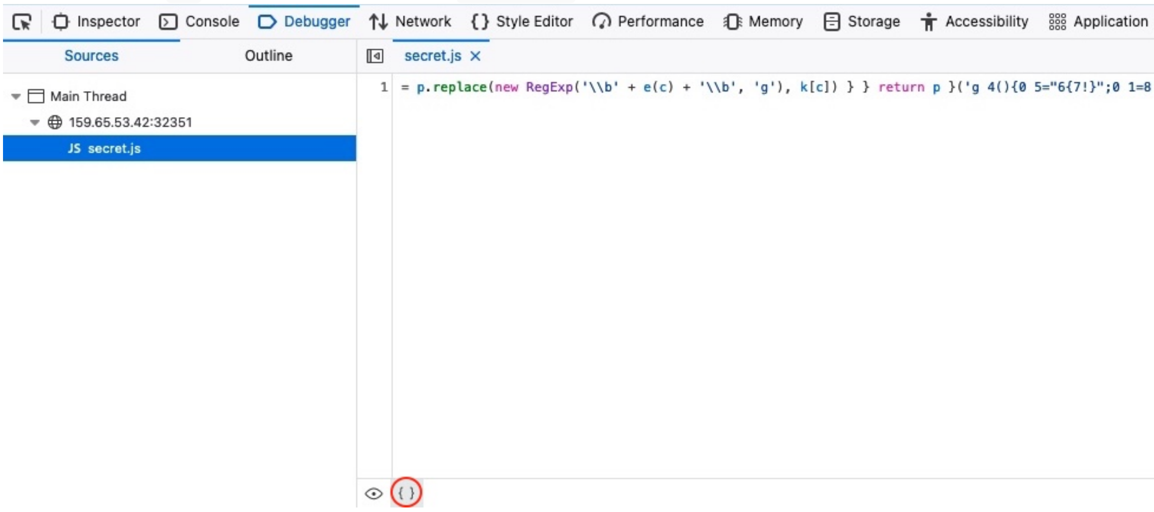
Deobfuscation

Now that we understand how code obfuscation works let's start our learning towards deobfuscation. Just as there are tools to obfuscate code automatically, there are tools to beautify and deobfuscate the code automatically.

Beautify

We see that the current code we have is all written in a single line. This is known as `Minified JavaScript` code. In order to properly format the code, we need to `Beautify` our code. The most basic method for doing so is through our `Browser Dev Tools`.

For example, if we were using Firefox, we can open the browser debugger with [`CTRL+SHIFT+Z`], and then click on our script `secret.js`. This will show the script in its original formatting, but we can click on the ' `{ }` ' button at the bottom, which will `Pretty Print` the script into its proper JavaScript formatting:



Furthermore, we can utilize many online tools or code editor plugins, like Prettier or Beautifier. Let us copy the `secret.js` script:

```
eval(function (p, a, c, k, e, d) { e = function (c) { return c.toString(36) }; if (!''.replace(/^/, String)) { while (c--) { d[c.toString(a)] = k[c] ||
c.toString(a) } k = [function (e) { return d[e] }]; e = function () { return '\\w+' }; c = 1 }; while (c--) { if (k[c]) { p = p.replace(new RegExp('\\b' +
e(c) + '\\b', 'g'), k[c]) } } return p }('g 4(){0 5="6{7!}";0 1=8 a();0 2="/9.c";1.d("e",2,f);1.b(3)}', 17, 17,
'var|xhr|url|null|generateSerial|flag|HTB|flag|new|serial|XMLHttpRequest|send|php|open|POST|true|function'.split('|'), 0, {}))
```

We can see that both websites do a good job in formatting the code:

Prettier v2.1.1

```
eval(function (p, a, c, k, e, d) { e = function (c) { return c.toString(36) }; if (!''.replace(
```

```
eval(
  (function (p, a, c, k, e, d) {
    e = function (c) {
      return c.toString(36);
    };
    if (!"".replace(/^/, String)) {
      while (c--) {
        d[c.toString(a)] = k[c] || c.toString(a);
      }
      k = [
        function (e) {
          return d[e];
        },
      ];
      e = function () {
        return "\\w+";
      };
      c = 1;
    }
```

```
eval(function (p, a, c, k, e, d) {
    e = function (c) {
        return c.toString(36)
    };
    if(!''.replace(/^/, String)) {
        while(c--) {
            d[c.toString(a)] = k[c] || c.toString(a)
        }
        k = [function (e) {
            return d[e]
        }];
        e = function () {
            return '\\w+'
        };
        c = 1
    };
    while(c--) {
        if(k[c]) {
            p = p.replace(new RegExp('\\b' + e(c) + '\\b', 'g'), k[c])
        }
    }
```

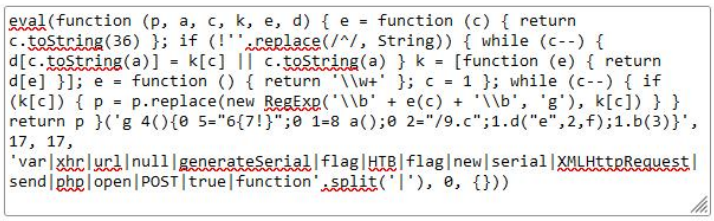
Beautify Code (ctrl-enter)

However, the code is still not very easy to read. This is because the code we are dealing with was not only minified but obfuscated as well. So, simply formatting or beautifying the code will not be enough. For that, we will require tools to deobfuscate the code.

Deobfuscate

We can find many good online tools to deobfuscate JavaScript code and turn it into something we can understand. One good tool is UnPacker. Let's try copying our above-obfuscated code and run it in UnPacker by clicking the `UnPack` button.

Tip: Ensure you do not leave any empty lines before the script, as it may affect the deobfuscation process and give inaccurate results.



We can see that this tool does a much better job in deobfuscating the JavaScript code and gave us an output we can understand:

```
function generateSerial() {
  ... SNIP ...
  var xhr = new XMLHttpRequest;
  var url = "/serial.php";
  xhr.open("POST", url, true);
  xhr.send(null);
};
```

As previously mentioned, the above-used method of obfuscation is `packing`. Another way of `unpacking` such code is to find the `return` value at the end and use `console.log` to print it instead of executing it.

Reverse Engineering

Though these tools are doing a good job so far in clearing up the code into something we can understand, once the code becomes more obfuscated and encoded, it would become much more difficult for automated tools to clean it up. This is especially true if the code was obfuscated using a custom obfuscation tool.

We would need to manually reverse engineer the code to understand how it was obfuscated and its functionality for such cases. If you are interested in knowing more about advanced JavaScript Deobfuscation and Reverse Engineering, you can check out the Secure Coding 101 module, which should thoroughly cover this topic.

Code Analysis

Now that we have deobfuscated the code, we can start going through it:

```
'use strict';
function generateSerial() {
  ... SNIP ...
  var xhr = new XMLHttpRequest;
  var url = "/serial.php";
  xhr.open("POST", url, true);
  xhr.send(null);
};
```

We see that the `secret.js` file contains only one function, `generateSerial`.

HTTP Requests

Let us look at each line of the `generateSerial` function.

Code Variables

The function starts by defining a variable `xhr`, which creates an object of `XMLHttpRequest`. As we may not know exactly what `XMLHttpRequest` does in JavaScript, let us Google `XMLHttpRequest` to see what it is used for.

After we read about it, we see that it is a JavaScript function that handles web requests.

The second variable defined is the `URL` variable, which contains a URL to `/serial.php`, which should be on the same domain, as no domain was specified.

Code Functions

Next, we see that `xhr.open` is used with `"POST"` and `URL`. We can Google this function once again, and we see that it opens the HTTP request defined ' `GET` or `POST` ' to the `URL`, and then the next line `xhr.send` would send the request.

So, all `generateSerial` is doing is simply sending a `POST` request to `/serial.php`, without including any `POST` data or retrieving anything in return.

The developers may have implemented this function whenever they need to generate a serial, like when clicking on a certain `Generate Serial` button, for example. However, since we did not see any similar HTML elements that generate serials, the developers must not have used this function yet and kept it for future use.

With the use of code deobfuscation and code analysis, we were able to uncover this function. We can now attempt to replicate its functionality to see if it is handled on the server-side when sending a `POST` request. If the function is enabled and handled on the server-side, we may uncover an unreleased functionality, which usually tends to have bugs and vulnerabilities within them.

HTTP Requests

In the previous section, we found out that the `secret.js` main function is sending an empty `POST` request to `/serial.php`. In this section, we will attempt to do the same using `cURL` to send a `POST` request to `/serial.php`. To learn more about `cURL` and web requests, you can check out the Web Requests module.

---

cURL

`cURL` is a powerful command-line tool used in Linux distributions, macOS, and even the latest Windows PowerShell versions. We can request any website by simply providing its URL, and we would get it in text-format, as follows:

```
curl http://SERVER_IP:PORT/

</html>
<!DOCTYPE html>

<head>
    <title>Secret Serial Generator</title>
    <style>
        *,
        html {
            margin: 0;
            padding: 0;
            border: 0;
...SNIP...
        <h1>Secret Serial Generator</h1>
        <p>This page generates secret serials!</p>
    </div>
</body>

</html>
```

This is the same `HTML` we went through when we checked the source code in the first section.

POST Request

To send a `POST` request, we should add the `-X POST` flag to our command, and it should send a `POST` request:

```
curl -s http://SERVER_IP:PORT/ -X POST
```

Tip: We add the "-s" flag to reduce cluttering the response with unnecessary data

However, `POST` request usually contains `POST` data. To send data, we can use the " `-d "param1=sample"` " flag and include our data for each parameter, as follows:

```
curl -s http://SERVER_IP:PORT/ -X POST -d "param1=sample"
```

Now that we know how to use `cURL` to send basic `POST` requests, in the next section, we will utilize this to replicate what `server.js` is doing to understand its purpose better.

Decoding

After doing the exercise in the previous section, we got a strange block of text that seems to be encoded:

```
curl http:/SERVER_IP:PORT/serial.php -X POST -d "param1=sample"

ZG8gdGhlIGV4ZXJjaXNlLCBkb24ndCBjb3B5IGFuZCBwYXN0ZSA7KQo=
```

This is another important aspect of obfuscation that we referred to in `More Obfuscation` in the `Advanced Obfuscation` section. Many techniques can further obfuscate the code and make it less readable by humans and less detectable by systems. For that reason, you will very often find obfuscated code containing encoded text blocks that get decoded upon execution. We will cover 3 of the most commonly used text encoding methods:

- `base64`
- `hex`
- `rot13`

Base64

`base64` encoding is usually used to reduce the use of special characters, as any characters encoded in `base64` would be represented in alpha-numeric characters, in addition to `+` and `/` only. Regardless of the input, even if it is in binary format, the resulting base64 encoded string would only use them.

Spotting Base64

`base64` encoded strings are easily spotted since they only contain alpha-numeric characters. However, the most distinctive feature of `base64` is its padding using `=` characters. The length of `base64` encoded strings has to be in a multiple of 4. If the resulting output is only 3 characters long, for example, an extra `=` is added as padding, and so on.

Base64 Encode

To encode any text into `base64` in Linux, we can echo it and pipe it with ' `|` ' to `base64`:

```
echo https://www.hackthebox.eu/ | base64

aHR0cHM6Ly93d3cuaGFja3RoZWJveC5ldS8K
```

Base64 Decode

If we want to decode any `base64` encoded string, we can use `base64 -d`, as follows:

```
echo aHR0cHM6Ly93d3cuaGFja3RoZWJveC5ldS8K | base64 -d

https://www.hackthebox.eu/
```

Hex

Another common encoding method is `hex` encoding, which encodes each character into its `hex` order in the `ASCII` table. For example, `a` is `61` in hex, `b` is `62`, `c` is `63`, and so on. You can find the full `ASCII` table in Linux using the `man ascii` command.

Spotting Hex

Any string encoded in `hex` would be comprised of hex characters only, which are 16 characters only: 0-9 and a-f. That makes spotting `hex` encoded strings just as easy as spotting `base64` encoded strings.

Hex Encode

To encode any string into `hex` in Linux, we can use the `xxd -p` command:

```
echo https://www.hackthebox.eu/ | xxd -p

68747470733a2f2f7777772e6861636b746865626f782e65752f0a
```

Hex Decode

To decode a `hex` encoded string, we can use the `xxd -p -r` command:

```
echo 68747470733a2f2f7777772e6861636b746865626f782e65752f0a | xxd -p -r

https://www.hackthebox.eu/
```

Caesar/Rot13

Another common -and very old- encoding technique is a Caesar cipher, which shifts each letter by a fixed number. For example, shifting by 1 character makes `a` become `b`, and `b` becomes `c`, and so on. Many variations of the Caesar cipher use a different number of shifts, the most common of which is `rot13`, which shifts each character 13 times forward.

Spotting Caesar/Rot13

Even though this encoding method makes any text looks random, it is still possible to spot it because each character is mapped to a specific character. For example, in `rot13`, `http://www` becomes `uggc://jjj`, which still holds some resemblances and may be recognized as such.

Rot13 Encode

There isn't a specific command in Linux to do `rot13` encoding. However, it is fairly easy to create our own command to do the character shifting:

```
echo https://www.hackthebox.eu/ | tr 'A-Za-z' 'N-ZA-Mn-za-m'

uggcf://jjj.unpxgurobk.rh/
```

Rot13 Decode

We can use the same previous command to decode rot13 as well:

```
echo uggcf://jjj.unpxgurobk.rh/ | tr 'A-Za-z' 'N-ZA-Mn-za-m'

https://www.hackthebox.eu/
```

Another option to encode/decode rot13 would be using an online tool, like rot13.

Other Types of Encoding

There are hundreds of other encoding methods we can find online. Even though these are the most common, sometimes we will come across other encoding methods, which may require some experience to identify and decode.

```
If you face any similar types of encoding, first try to determine the type of encoding, and then look for online tools to decode it.
```

Some tools can help us automatically determine the type of encoding, like Cipher Identifier. Try the encoded strings above with Cipher Identifier, to see if it can correctly identify the encoding method.

Other than encoding, many obfuscation tools utilize encryption, which is encoding a string using a key, which may make the obfuscated code very difficult to reverse engineer and deobfuscate, especially if the decryption key is not stored within the script itself.

Skills Assessment

During our Penetration Test, we came across a web server that contains JavaScript and APIs. We need to determine their functionality to understand how it can negatively affect our customer.

Summary

Congratulations on reaching the end of `JavaScript Deobfuscation`. We hope you can now recognize obfuscated scripts and deobfuscate them, decode their output, and replicate their functions.

The following is a summary of what we learned:

- First, we uncovered the `HTML` source code of the webpage and located the JavaScript code.
- Then, we learned about various ways to obfuscate JavaScript code.
- After that, we learned how to beautify and deobfuscated minified and obfuscated JavaScript code.
- Next, we went through the deobfuscated code and analyzed its main function
- We then learned about HTTP requests and were able to replicate the main function of the obfuscated JavaScript code.
- Finally, we learned about various methods to encode and decode strings