Written by Stefan Judis

Published at
Jul 16 2017

Updated at
Jun 30 2018

Reading time
7min

Views
1976



100% agent accuracy with steering hooks. Learn how.

ADS VIA CARBON

Recently I came across this tweet by @FakeUnicode . It included a JavaScript snippet which looked pretty harmless but resulted in a hidden message being alerted. I took me a while to understand what's going on so I thought that documenting the steps I took could be interesting to someone.

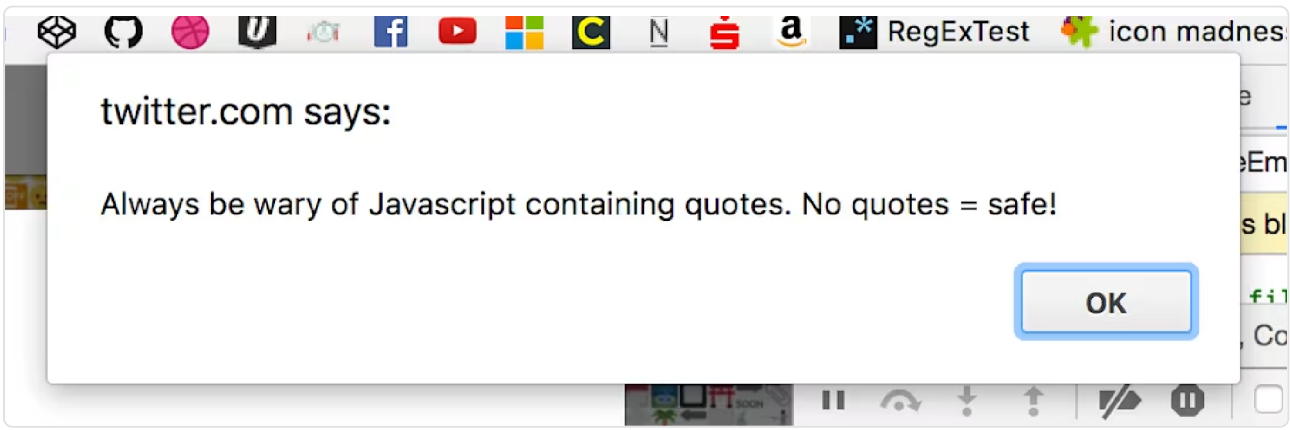
The snippet was the following:

JS

```
for(A in {A:0}){
  alert(unescape(escape(A).replace(/u.{8}/g,[])))
};
```

So, what are you expecting to happen here?

It uses a `for in` loop which iterates over enumerable properties of an object. There is only the property `A` in it so I thought it's an alert showing up presenting the letter `A`. Well... I was wrong. :D



This surprised me and I started debugging using the Chrome console.

The discovery of hidden code points

The first thing I did was simplifying the snippet to see what's going on.

JS

```
for(A in {A:0}){console.log(A)};
// A
```

Hmm... okay nothing going on here. So I continued.

JS

```
for(A in {A:0}){console.log(escape(A))};
// A%uDB40%uDD6C%uDB40%uDD77%uDB40%uDD61%uDB40%uDD79%uDB
```

Holy! Where's all this coming from?

So I took a step back and had a look at the length of the string.

JS

```
for(A in {A:0}){console.log(A.length)};
// 129
```

Interesting. Next, I copied the `A` from the object and already discovered that the Chrome console was dealing with something hidden here because the cursor was "stuck" for several left/right key strokes.

But let's have a look at what's in there and the get the values of all the 129 code units:

JS

```
const propertyName = 'A';
for(let i = 0; i < propertyName.length; i++) {
  console.log(propertyName[i]);
  // to get code unit values use charCodeAt
  console.log(propertyName.charCodeAt(i));
}
// A
// 65
// 
// 56128
// 
```

```
// 56684
// ...
```

What you see there is the letter `A` which has the code unit value `65` followed by several code units somewhere around 55 and 56 thousand which by `console.log` are displayed with the well-known question mark meaning that the system doesn't know how to handle this code unit.

Surrogate pairs in JavaScript

These values are parts of so called **surrogate pairs** which are used to represent code points that have a value bigger than 16 bit (or in other words have a code point value bigger than `65536`). This is needed because Unicode itself defines 1,114,112 different code points and the string format used by JavaScript is UTF-16. This means that only the first 65536 code points defined in Unicode can be represented in a single code unit in JavaScript.

A bigger value can then be evaluated by applying a crazy formula to the pair which results then in a value being bigger than `65536`.

*Shameless plug: I give **a talk** on exactly this topic which might help you understand concepts of code points, emojis, and surrogate pairs.*

So what we discovered were 129 code units of which 128 are surrogate pairs representing 64 code points. So what are these code points?

To retrieve code point values from a string there is the really handy `for of` loop which iterates over string code points (and not over code units like the first `for` loop) and also the `...` operator which uses `for of` under the hood.

JS

```
console.log([...'A']);
// (65) ["A", "", "", "", "", "", "", "", "", "", ""
```

So, `console.log` doesn't even know how to display these resulting code points so let's check what we're dealing with in detail.

JS

```
// to get code point values use codePointAt
console.log([...'A'].map(c => c.codePointAt(0)));
// [65, 917868, 917879, ...]
```

Side note: be aware that there are two different functions when dealing with code units and code points in JavaScript 👉 charCodeAt and codePointAt . They behave slightly different so you might want to have a look.

Identifiers names in JavaScript objects

The code points `917868`, `917879` and the following are part of the Variation Selectors Supplement in Unicode. Variation selectors in Unicode are used to specify standardized variation sequences for mathematical symbols, emoji symbols, 'Phags-pa letters, and CJK unified ideographs corresponding to CJK compatibility ideographs. These are usually not meant to be used alone.

Okay cool, but why does this matter?

When you head over to the ECMAScript spec you find out that property identifier names can include more than just "normal characters".

```
Identifier ::
  IdentifierName but not ReservedWord
IdentifierName ::
  IdentifierStart
  IdentifierName IdentifierPart
IdentifierStart ::
  UnicodeLetter
  $
  _
  \ UnicodeEscapeSequence
IdentifierPart ::
  IdentifierStart
  UnicodeCombiningMark
  UnicodeDigit
  UnicodeConnectorPunctuation
  <ZWNJ>
  <ZWJ>
```

So what you see above is that an identifier can consist of an `IdentifierName` and an `IdentifierPart`. The important part is the definition for `IdentifierPart`. As long as it's not the first character of an identifier the following identifier names are completely valid:

JS

```
const examples = {
  // UnicodeCombiningMark example
  somethingî: 'LATIN SMALL LETTER I WITH CIRCUMFLEX',
  somethingi\u0302: 'I + COMBINING CIRCUMFLEX ACCENT',

  // UnicodeDigit example
  something৷: 'ARABIC-INDIC DIGIT ONE',
  something\u0661: 'ARABIC-INDIC DIGIT ONE',

  // UnicodeConnectorPunctuation example
  something‿: 'DASHED LOW LINE',
```

```
  something\ufe4d: 'DASHED LOW LINE',

  // ZWJ and ZWNJ example
  something\u200c: 'ZERO WIDTH NON JOINER',
  something\u200d: 'ZERO WIDTH JOINER'
}
```

So when you evaluate this expression you get the following result

```
{
  somethingî: "ARABIC-INDIC DIGIT ONE",
  somethingî: "I + COMBINING CIRCUMFLEX ACCENT",
  something\: "ARABIC-INDIC DIGIT ONE"
  something_: "DASHED LOW LINE",
  something: "ZERO-WIDTH NON-JOINER",
  something: "ZERO-WIDTH JOINER"
}
```

This brings me to my learning of the day. 🎉

According to the ECMAScript spec:

> *Two IdentifierName that are canonically equivalent according to the Unicode standard are not equal unless they are represented by the exact same sequence of code units.*

This means the two object identifier keys can look exactly the same but consist of different code units and this means that they'll be both included in the object. Like in this case `î` which has the code unit value `00ee` and the character `i` with a trailing `COMBINING CIRCUMFLEX ACCENT`. So they're not the same and it looks like you've got doubled properties included in your object. The same goes for the keys with a trailing [Zero-

Width joiner or Zero-Width non-joiner . They look the same but are not!

But back to topic: the Variation Selectors Supplement values we found belong to the `UnicodeCombiningMark` category which makes them a valid identifier name (even when they're not visible). They're invisible because most likely the system only shows the result of them when used in a valid combination.

The escape function and some string replacement

So what the `escape` function does is that it goes over all the code units and escapes every unit . This means it grabs the starting letter `A` and all the parts of the surrogate pairs and simply transforms them to strings again. The values that don't have been visible will be "stringified". This is the long sequence you saw in the beginning of the article.

```
A%uDB40%uDD6C%uDB40%uDD77%uDB40%uDD61%uDB40%uDD79%uDB40%
```

The trick now is that @FakeUnicode picked specific Variation Selectors namely the ones that end with a number that maps back to an actual character. Let's look at an example.

JS

```
// a valid surrogate pair sequence
'%uDB40%uDD6C'.replace(/u.{8}/g,[]);
// %6C 👉 6C (hex) === 108 (dec) 👉 LATIN SMALL LETTER L
unescape('%6C')
// 'l'
```

One thing that looks a bit cryptic is that the example uses an empty array `[]` as string replacement value which will be evaluated using

`toString()` which means that it evaluates to `''`.

An empty string does the job, too. The reasoning for going with `[]` is that this way you can bypass quotes filter or something similar .

This way it's possible to encode a whole message with invisible characters.

Overall functionality

So when we look at this example again:

```JS
for(A in {A:0}){
  alert(unescape(escape(A).replace(/u.{8}/g,[])))
};
```

What happens is:

- `A:0` - the `A` includes a lot of "hidden code units"
- these characters become visible using `escape`
- a mapping is performed using `replace`
- the mapping result will be unescaped again to be displayed in the alert window

I think this is pretty cool stuff!

Additional resources

This little example covers a lot of Unicode topics. So if you want to read more I highly recommend you to read Mathias Bynens articles on Unicode and JavaScript:

- JavaScript has a Unicode problem
- JavaScript character escape sequences

Reply to this post and share your thoughts via good old email.



About Stefan Judis

Frontend nerd with over ten years of experience, freelance dev, "Today I Learned" blogger, conference speaker, and Open Source maintainer.

Related Topics

JavaScript

Related Articles

- Escape from Average
- Intl can localize units, too!
- New lines are removed from WHATWG URLs
- Feature detection of View Transition Types
- Stringly Typed

Topics

JavaScript 132 CSS 107 Tools 64 HTML 56

Bash 45 NodeJS 42 Accessibility 38 Web 30

git 23 VSCode 21 macOS 20 Performance 20

Random Web Dev Learnings

Suffix aliases (-s) in Zsh

[is an actual bash command

How to visualize the tab order without using an extension in Firefox

em and strong are not announced by screen readers

includes really is the better indexOf

New lines are removed from WHATWG URLs

Popular articles

em and strong are not announced by screen readers

There is an MDN short URL to access the docs

How to animate an element's height with CSS grid

Recently updated resources

Staying up to date

Writing good docs, blogs and technical articles

Public speaking

stefanjudis.com © 2026 by Stefan Judis is licensed under CC BY-NC-SA 4.0

All rights reserved and content created without AI.