SANS Institute
Information Security Reading Room

Developing a JavaScript Deobfuscator in .NET

Roberto Nardella

Copyright SANS Institute 2021. Author Retains Full Rights.

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

https://t.me/learningnets

Developing a JavaScript Deobfuscator in .NET

GIAC (GREM) Gold Certification

Author: Roberto Nardella
Advisor: *Lenny Zeltser*

Accepted: *October 16th, 2020*

Abstract

JavaScript, a core technology of the World Wide Web, is a recently born scripting language and, starting from its early years, became notorious within the cyber security community not only for well-known security problems like Cross Site Scripting (XSS) or Cross Site Request Forgery (CSRF), but also for its flexibility in offering a valid vehicle for the implementation of the first stage of a malware attack.
The first phase of a malware infection chain is very often a dropper that uses obfuscated script code, as this is an efficient means to evade AV/IDS detection and to complicate static analysis.
Although several efficient tools assisting the de-obfuscation process are already available on the Internet, either as open/closed source software or online web-based ones, there is certainly room for the development of new tools, and the .NET Framework offers a wide plethora of very useful functionalities.
This paper explores the capabilities of the .NET Framework for de-obfuscating JavaScript scripts, either using .NET Framework built-in APIs or third party packages for .NET, explaining their usage in the context of JavaScript de-obfuscator development. The research will be carried out by merging several inputs from two perspectives: the programming world and the script reverse engineering world.

© 2020 The SANS Institute, Author Retains Full Rights

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

1. Introduction

"ECMAScript" is a programming language and, at the same time, a standardized specification developed by computer technologist Brendan Eich of Netscape. Based on these specifications (described in a document named "ECMA-262"), the birth of the "JavaScript language" was publicly announced by Netscape and Sun Microsystems in a press release in 1995 (Krill, 2008): "JavaScript" and "ECMAScript" are essentially two different names for the same scripting language, although the term "JavaScript" is the most commonly used term.

Using script code (JavaScript or VBScript) as the first stage of a malware infection is a method that allows malware developers to bypass defenses more easily than dropping a single binary containing the whole malicious logic. JavaScript obfuscation is much easier to implement than binary obfuscation and, due to the dynamicity of the scripting language, writing a signature that detects the so-called "malware droppers" is a difficult task. From an incident responder's standpoint, it is very important to reverse engineer said obfuscated scripts, as they contain very useful indicators ranging from the Command and Control (C2) servers used by threat actors, executables within the impacted operating system used by the malicious script itself, and also second stage malware payloads, or payload parts.

Several tools assist malware analysts in the de-obfuscation process; many of them are online resources while other tools are available in the form of compiled, closed-source software or open-source scripts. A first constraint for an analyst working in a corporate environment is that online web-based de-obfuscators cannot be used because of corporate policies. If those scripts would contain hardcoded (and obfuscated) information regarding the target, that information would be delivered to unknown hands, with almost always no possibility of deleting it. The dynamicity of obfuscation patterns changing frequently is another characteristic that makes automation and tool development an important aspect in the life of an analyst. Often, reverse engineers need to create tools "on the fly" to compensate for the lack of availability of tools that perform very specific analyses so it is important to explore the possibilities that programming languages can offer. Analysts often prefer to use tools that do not require the installation of numerous dependencies while other analysts may find GUI-based tools more practical for the purpose of de-obfuscation. This is because malicious code often needs to be modified in order to facilitate its de-obfuscation. These are additional

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

reasons why another approach will be explored, and a new tool will developed for this research.

1.1. Why Choose .NET?

The .NET Framework and, in particular, the C# language was chosen for this project because of the following usability criteria: in general, researchers and software users may find using closed-source, compiled software very practical because they usually come with necessary libraries already bundled in the installer, or in the ZIP file containing the entire solution. Some tools coded in scripting languages like Python, Perl or PowerShell require dependencies that the user needs to download, install and properly configure.

The second reason why C# was the chosen programming language is due to the fact that, compared to other compiled languages like C or C++, it is safer and less complex. In addition, the .NET Framework offers a comprehensive API already contained within it. The Microsoft Visual Studio IDE facilitates the development of GUI-based applications through a quick “What You See Is What You Get” (WYSIWYG) wizard.

2. Research Methodology

The methodology used for this research consisted of selecting API and code snippets from either the Microsoft.NET Framework or third party packages for .NET, using said code to build a JavaScript de-obfuscation application with Microsoft Visual Studio 2017 environment (C# language) and finally testing the obtained software against malicious JavaScript code. The development of JavaScript de-obfuscation software is directly related to the purpose of this research, which is to explore the .NET Framework’s capabilities and to demonstrate what is possible, and conversely, to identify what's impractical. Building an application is essential to demonstrate whether .NET capabilities are concretely useful in the

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.
© 2021 The SANS Institute, Author Retains Full Rights

context of malicious JavaScript analysis. The intended functionalities for this application can be divided into three main categories:

1) "Decoders": functions that reverse encoded text (Base64, Hex, escaped Unicode, and so on) into clear text;
2) "Evaluators": functions that dynamically de-obfuscate a relatively larger amount of code by executing it and returning a more human readable output;
3) "AST evaluation": functions that perform a static, or partially static de-obfuscation by converting selected JavaScript code snippets into their corresponding AST (Abstract Syntax Tree), then dynamically evaluate some of its elements ("partial evaluation"), manipulating the AST and finally reverting back from AST into JavaScript code.

For the development of this application, several alternative evaluators will be tested, and their obtained outputs will be compared to each other. Tests will be run by deploying the obtained app in Windows 7 and Windows 10 operating system environments.

The malicious JavaScript code chosen to test the application was selected from the "Geeks on Security" public GitHub repository (n.d., "Geeks on Security, 2020): this repository, other than storing a great number of malicious script samples, also contains malware samples in either their original obfuscated version or in the de-obfuscated one. Selecting and using these specific samples for the test is particularly useful since the results obtained from the software testing can be easily compared with the de-obfuscated version of these samples in the GitHub repository.

3. Obfuscated JavaScript

Code obfuscation is an intentional transformation of the code in order to make it more difficult to reuse or analyze and, as mentioned, it is a practice used for either legitimate purposes (protecting intellectual property) or malicious purposes (IDS evasion). A Master's Thesis from computer scientist Alejandro Pardo Lopez , Radboud University of Nijmegen (The Netherlands) entitled "Seek And De-obfuscate - Uncovering JavaScript" (2008) and a

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

research entitled “The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study”, published by Pennsylvania University researchers Wei Xu, Fangfang Zhang and Sencun Zhu (2012) explain the art of JavaScript obfuscation in greater detail. In particular, this latter research paper enumerates the most common JavaScript obfuscation patterns:

1. Encoding: two ways of encoding are shown in the publication; the first way is to convert the code into escaped ASCII characters, Unicode or hexadecimal representations. The second method uses customized encoding functions, where attackers usually use an encoding function to create the obfuscated code and attach a decoding function to decode it during execution;
2. String splitting / number splitting: consists of converting a string into a concatenation of several substrings. In the case of numbers, the splitting consists of replacing the number with a mathematical expression returning that number as a result;
3. Randomization obfuscation: in this type of obfuscation, some elements of JavaScript code are inserted or changed without changing the semantics of the code. In this type of obfuscation, some elements of JavaScript codes are inserted or changed without changing the semantics of the original code. Common techniques used in this category are white space randomization, comment randomization, and variable and name randomization;
4. Logic Obfuscation: manipulate the execution paths of JavaScript codes by changing the logic structure, without affecting the original semantics. There are two ways to implement logic structure obfuscation. One way is to insert some instructions which are independent of the functionality. The other one is to add or change some conditional branches (Xu, Zhang, Zhu, 2012).

4. The GUI: Scintilla.NET

The first element that was incorporated in the tool developed for this research is a control named “Scintilla” (Hodgson, 2002) which is a Windows Forms control for text

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

editing, and also the core library for the well-known editor "Notepad++". In particular, the .NET port ("Scintilla.NET"), created by developer Jacob Slusser (2018) was used. "Notepad++" is a very handy tool used by developers to edit their source code, and JavaScript de-obfuscation is an activity that often requires manual code tweaking to ensure proper code evaluation. In addition, some features like syntax highlighting, code folding, multi-selection and brace matching can help with better handling and visualization of the code, as shown in Figure 1.

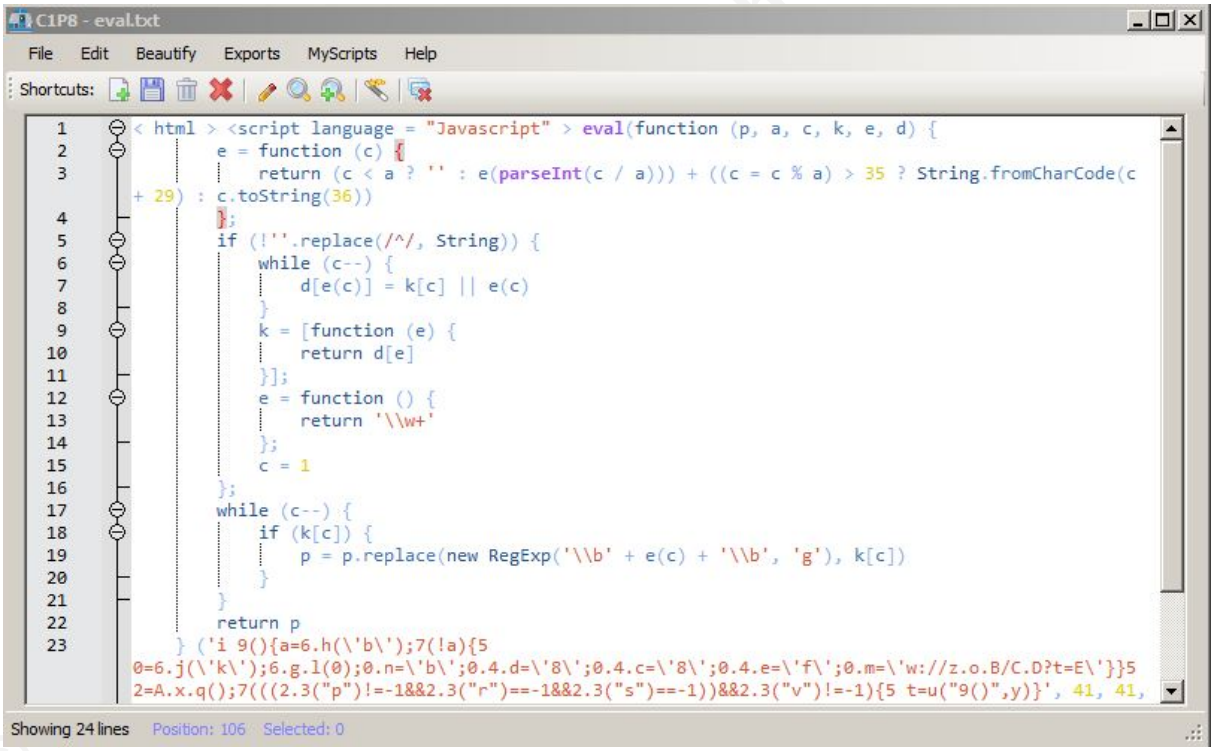

Figure 1 - a GUI based application using Scintilla.NET

In particular, for the development of this specific application, the "selectedText" property and the "ReplaceSelection" method of the Scintilla control (in Table 1 below, instantiated as "scintilla1") were widely used in order to allow greater interactivity with the user, who can then apply a given de-obfuscation method to manually selected text only:

```
string alfa = scintilla1.SelectedText;
string beta = WhateverMethod(alfa);
scintilla1.ReplaceSelection(beta);
```

Table 1 - Scintilla.NET methods for selected text

Author Name, email@address

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

Another useful component used for the development of this tool is the Scintilla' "FindReplace" dialog control, a fork of the Scintilla project created by developer "Stumpii" (Towner, 2017) which provides a "Find Replace" dialog box very similar to the "Notepad++" one. Given its integration with a regular expression engine and other sophisticated find and replace features, this represents a valid aid for all the cases where source code manipulation is needed, during the JavaScript de-obfuscation process.

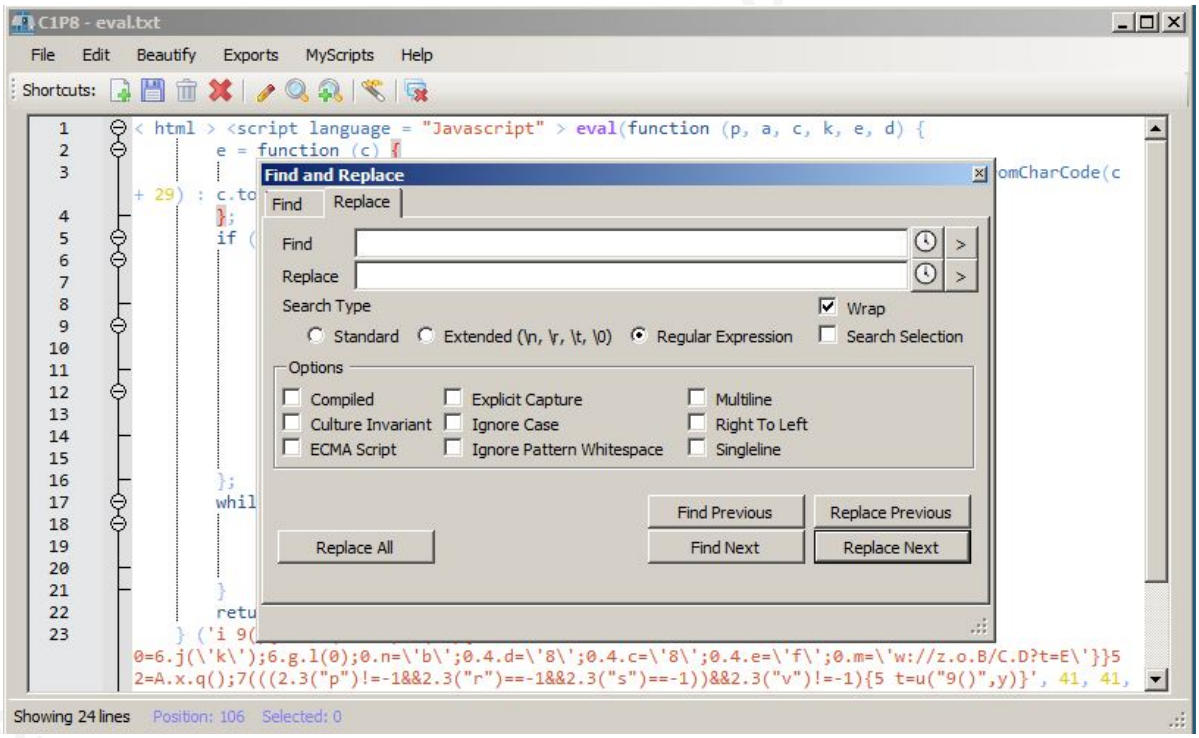

Figure 2 - Scintilla.NET "FindReplaceDialog" control

The Scintilla.NET "FindReplaceDialog" control offers an overload of its "ReplaceAll" method which accepts "search flags" as one of its arguments. The "WholeWord" property of said flags checks for word boundaries—the code in Table 2 below will take the string that a user selects in the GUI with the mouse, conduct a search through the entire opened document and replace it with string "AAA". In the case the user selected text is "int", this method will replace "int" with "AAA" but will leave all the words containing "int" (for example: "integer") intact. This is useful when dealing with "Randomization Obfuscation" patterns, as

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

mentioned in Section 3, and in particular to assign more meaningful names to variables and functions of the obfuscated code.

```
using ScintillaNET;
using ScintillaNET_FindReplaceDialog;
[….]
     string Str = "AAA";
     FindReplace gino = new FindReplace(scintilla1);
     int count = gino.ReplaceAll(scintilla1.SelectedText, Str,
SearchFlags.WholeWord, false, false);
```

Table 2 - Scintilla "FindReplaceDialog" method "ReplaceAll"

5. Decoders in .NET

As mentioned at section 3, one of the most common obfuscation means is "encoding", which is the transformation of clear text according to a given encoding standard. The .NET Framework offers numerous methods that can decode the most common encoding patterns.

5.1. Base64

Base64 is a group of binary-to-text encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation. The term Base64 originates from a specific MIME content transfer encoding. Base64 decoding methods are already implemented in .NET under the "System.Convert" namespace. The following three code blocks convert user-selected text from Base64 respectively into ASCII, Unicode and UTF8 format:

```
     string alfa = scintilla1.SelectedText;
     byte[] data = System.Convert.FromBase64String(alfa);
     string beta = Encoding.ASCII.GetString(data);
     scintilla1.ReplaceSelection(beta);

     string alfa = scintilla1.SelectedText;
     byte[] data = System.Convert.FromBase64String(alfa);
     string beta = Encoding.Unicode.GetString(data);
     scintilla1.ReplaceSelection(beta)

     string alfa = scintilla1.SelectedText;
     byte[] data = System.Convert.FromBase64String(alfa);
     string beta = Encoding.UTF8.GetString(data);
     scintilla1.ReplaceSelection(beta);
```

Table 3 - .NET Framework API for Base64 decoding

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

5.2. Unescape

"Escaped characters" are an alternative representation of characters that use their corresponding hexadecimal, octal or ASCII value. This representation, needed to reference special elements of a text like "carriage return" or "new line", is commonly found in obfuscated malicious scripts as it is an efficient means to encode strings. The "backslash character" ("\") is referred to as "escape character" as it is a standard notation to represent such escaped characters. The namespace "System.Text.RegularExpressions.Regex" offers a powerful "Unescape" method that can decode escaped strings in a great variety of escape notations:

```
private void AnyFunction(object sender, EventArgs e)
     {
      string alfa = scintilla1.SelectedText;
      string beta = alfa.Replace("%", "\\");
      string gamma = System.Text.RegularExpressions.Regex.Unescape(beta);
      scintilla1.ReplaceSelection(gamma);
     }
```

Table 4 - .NET Framework API for characters sequence unescape

5.3. Hex to char conversion

Hex to char conversion is the process of transforming a sequence of hexadecimal characters (for example: "\x73\x61\x6d\x70\x6c\x65" or "0x73 0x61 0x6d 0x70 0x6c 0x65", or "73 61 6d 70 6c 65") into its ASCII equivalent. The previously seen "Unescape" method already covers the decoding of text represented into hex format. However, there are different conventions to represent hexadecimal values. The "Int16.Parse()" method is provided by the core of the .NET Framework (namespace: "System") . One of the overloads of this method accepts the "NumberStyles" enum (from the "System.Globalization" namespace) which helps recognize different notation patterns for hexadecimal sequences:

```
private void AnyFunction(object sender, EventArgs e)
{

    StringBuilder builder = new StringBuilder();
    foreach (string elemento in gruppi)
```


© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

```
           {
            builder.Append((char)Int16.Parse(elemento,
NumberStyles.AllowHexSpecifier));
                    builder.Append(" ");
            }

       string risultato = builder.ToString();
       scintilla1.ReplaceSelection(risultato);
            }
```

Table 5 - .NET Framework API for Hex conversions

5.4. HTML entities decode

"HTML entities" are not an encoding mechanism but a representation of HTML reserved characters. Very often, obfuscated malicious JavaScript code reaches the target via HTML pages (of compromised websites, for example) and interacts with, and modifies, the HTML code of the page the malicious script is embedded into. The capability of translating an "&" character sequence into "&", or converting "<" and "&#60" into the "less than sign" ("<") is offered by the "HtmlDecode" method of the "System.Net.WebUtility" namespace. Even in the case of this function, its usage is very simple and straightforward:

```
            private void AnyFunction(object sender, EventArgs e)
    {
        string alfa = scintilla1.SelectedText;
        string beta = System.Net.WebUtility.HtmlDecode(alfa);
        scintilla1.ReplaceSelection(beta);
    }               string risultato = builder.ToString();
        scintilla1.ReplaceSelection(risultato);
        }
```

Table 6 - Utility for HTML entities decoding

5.5. URL Decoding

"Url Encoding" is another common encoding mechanism that is specific to URLs. This encoding standard was implemented to transmit data over the internet but is commonly found in obfuscated JavaScript, as it is still used to evade detection of blacklisted URLs. This encoding method, also known as "percent encoding", consists of representing a URL by translating its reserved and non-ascii characters in UTF8 format (for example: translating "https://www.google.it" into "https%3A%2F%2Fwww.google.it%2F"). "Full URL

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

encoding" patterns are also common to be found in malicious scripts (for example, "https://www.google.it" into "%68%74%74%70%73%3a%2f%2f%77%77%77%2e%67%6f%6f%67%6c%65%2e%69%74"). Other than the aforementioned "HtmlDecode" method, the "System.Net.WebUtility" namespace also provides the "UrlDecode" method. An advantage of using this method along with all the previously mentioned ones is that a "full URL encoding" pattern can be translated with other functions (for example, the "Int16.Parse()" method, or the "Unescape()" method). This latter method has the capability of recognizing the ASCII or non-reserved characters within "https%3A%2F%2Fwww.google.it%2F" and leaves them intact, while translating encoded characters only:

```
        private void AnyFunction(object sender, EventArgs e)
        {
            string alfa = scintilla1.SelectedText;
            string beta = System.Net.WebUtility.UrlDecode(alfa);
            scintilla1.ReplaceSelection(beta);
        }
```

Table 7 - Utility for URL decoding

Once all these methods are implemented in the application, a context menu connected to the Scintilla control of the graphical user interface provides a fast user interaction by calling these decoding methods with a simple right-click mouse operation, as seen in the below screenshot:

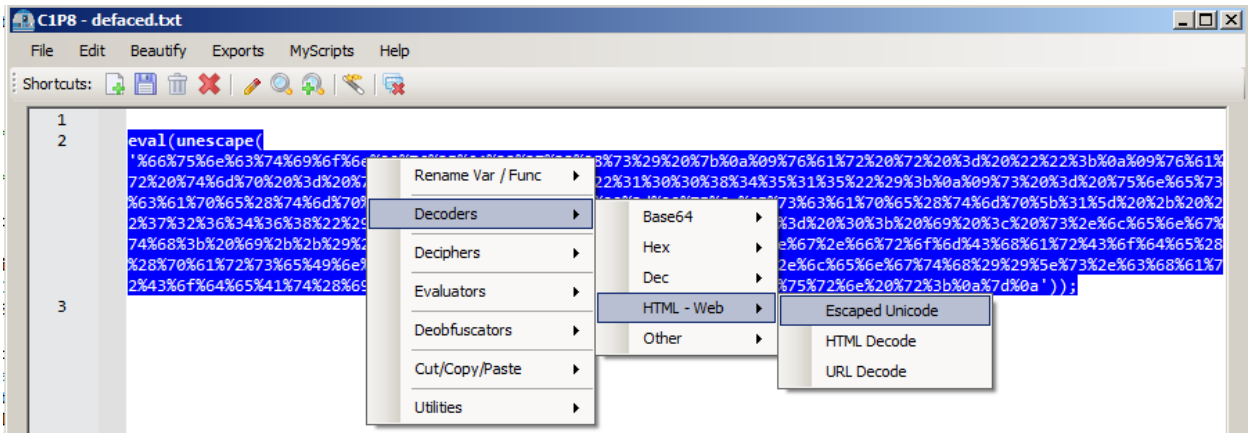

Figure 3 - Decoders implemented in the GUI of the application

Author Name, email@address

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

6. JavaScript Engines in .NET

"Code evaluation" is a dynamic method used in the de-obfuscation process which consists in running source code with a dedicated engine or interpreter to obtain transformed code. This can be achieved in two ways: the first is by implementing a JavaScript engine within the application and the second is using third party interpreters (like "SpiderMonkey", or "Cscript"). Also, the GUI can be used as a wrapper to run these external tools, capture their standard output and display it in the main GUI control. For this second option, The .NET Framework offers a "Process" class (from namespaces: "System.Diagnostics" and "System.ComponentModel") which are ideal for this kind of implementation. Testing "SpiderMonkey" and "Cscript" is out of the scope of this paper; nevertheless, these two applications will be embedded in the software that will be created for this research, with the obtained application also acting as a wrapper for these two tools, amongst other functionalities. For this research, JavaScript engines coming from the .NET Framework of from third party packages, available via Nuget, will be tested, and test results will be described in the next paragraphs. Three of the most popular JavaScript engines for .Net currently are:

1. VSA (.NET JScript engine, deprecated as of year 2005);
2. Jint (JavaScript engine from programmer Sebastien Ros);
3. Clearscript V8 (.NET porting of the popular Chrome engine);

The complexity of the source code needed to implement a JavaScript engine within an application may vary. Additionally, the results that can be obtained with each engine may differ—for these reasons, all the three engines in the above list will be tested.

6.1. Microsoft VSA (Jscript)

Developer Rick Strahl, in a post on his blog dated February 2007, shows a very straightforward way to implement a Microsoft Jscript built-in engine named "VSA". At the time this post was published, the "VSA Engine" was not obsolete as it is today—compiling source code incorporating this engine in Visual Studio will return a warning message "'Microsoft.JScript.Vsa.VsaEngine' is obsolete: 'Use of this type is not recommended because

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

it is being deprecated in Visual Studio 2005". Nevertheless, the code still compiles correctly and VSA returns good results. The code snippet below, which is a simplified version of the code that Strahl published, is an implementation of the VSA engine that takes source code as input and returns evaluated code to the caller:

```
public static Microsoft.JScript.Vsa.VsaEngine Engine =
Microsoft.JScript.Vsa.VsaEngine.CreateEngine();

public static object EvalJScript(string JScript) {
 object Result = null;
 try {
     Result = Microsoft.JScript.Eval.JScriptEvaluate(JScript, Engine);
 }
 catch () {            }
 return Result;
}
```

Table 8 - Source code for implementing the JScript VSA engine

After implementing the VSA engine in the application developed for this research, preliminary tests to verify its efficiency were carried out. The initial outcome is that the implementation of this engine using the code at Table 8 is not sufficient to evaluate a full malicious source code, however it can be used to process small code snippets and evaluate math expressions within the source code, concatenate strings, and resolve basic JavaScript native functions (like "Unescape()", "charAt()", and so on). The first limitation of this engine is its lack of support of the W3C DOM (Document Object Model) objects. Objects like "Document" (representing the body of an HTML document) or "Window" (representing the window of a web browser) are specific of web browsers and they are not understood by the VSA engine, since neither this engine nor the Scintilla control contain definitions for these objects. Malware researcher Lenny Zeltser developed a script, named "objects.js" (2016) which, in the context of a JavaScript de-obfuscation using tools "SpiderMonkey" or "Cscript", assists with emulating the behavior of a web browser or a PDF viewer since said object definitions are contained in the "objects.js" script itself. Many of the object's functions defined within "objects.js" use the native "Print" command of the SpiderMonkey interpreter to return de-obfuscated output as standard output, instead of executing it. Unfortunately, the

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

VSA browser does not have a built-in “Print” function or a “Wscript.echo” method (such as the “Cscript” Windows scripting host does). Nevertheless, this problem can be easily solved with the usage of the “.ToString()” native method in a similarly defined code snippet:

```
var document = {
    write: function(x){
    x.toString();
 },

writeln: function(x){
    x.toString();
    }
};
```

Table 9 - JavaScript code for object definition for the VSA engine

Another limitation of the VSA engine lies in the fact that it supports ECMAScript Version 6. Starting from ECMAScript Version 5, the so-called “strict mode” is adopted (MDN Web Docs, 2020); the inference in the context of JavaScript de-obfuscation is that the evaluation of a variable assignment (for example: “a = 12;”) will return an error message (“variable ‘a’ is not declared”) which will require tweaking the code in order to evaluate it correctly.

6.2. Jint

“Jint” is a JavaScript interpreter released by developer Sebastien Ros (2013). The source code for the implementation of Jint is as simple as the source code needed for the implementation of the VSA engine:

```
string beta = scintilla1.SelectedText;
var engine = new Jint.Engine();
var result = engine.Execute(beta).GetCompletionValue();
scintilla1.ReplaceSelection(result.ToString());
```

Table 10 - Source code for the implementation of the JINT JavaScript engine

Similar to the VSA engine, Jint does not have defined objects like “Document” or “Window” , so a definition similar to the one at Table 9 is needed. Similar to the VSA evaluator shown at paragraph 6.1, Jint will evaluate small code blocks and execute math

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

operations, string concatenation and native JavaScript functions such as “Unescape()”. In order to compare evaluation results between VSA and Jint, the following JavaScript code was used:

```
unescape('%66%75%6e%63%74%69%6f%6e%20%76%35%64%33%37%33%28%73%29%20%7b%0a%09%76%61%72%20%72%20%3d%20%22%22%3b%0a%09%76%61%72%20%74%6d%70%20%3d%20%73%2e%73%70%6c%69%74%28%22%31%30%30%38%34%35%31%35%22%29%3b%0a%09%73%20%3d%20%75%6e%65%73%63%61%70%65%28%74%6d%70%5b%30%5d%29%3b%0a%09%6b%20%3d%20%75%6e%65%73%63%61%70%65%28%74%6d%70%5b%31%5d%20%2b%20%22%37%32%36%34%36%38%22%29%3b%0a%09%66%6f%72%28%20%76%61%72%20%69%20%3d%20%30%3b%20%69%20%3c%20%73%2e%6c%65%6e%67%74%68%3b%20%69%2b%2b%29%20%7b%0a%09%09%72%20%2b%3d%20%53%74%72%69%6e%67%2e%66%72%6f%6d%43%68%61%72%43%6f%64%65%28%28%70%61%72%73%65%49%6e%74%28%6b%2e%63%68%61%72%41%74%28%69%25%6b%2e%6c%65%6e%67%74%68%29%29%5e%73%2e%63%68%61%72%43%6f%64%65%41%74%28%69%29%29%2b%37%29%3b%0a%09%7d%0a%09%72%65%74%75%72%6e%20%72%3b%0a%7d%0a');
```

Table 11 - "Unescape" JavaScript native function used for testing

The first result was that, while VSA evaluates this “Unescape()” function correctly, Jint does not interpret the line feed ASCII value “0a”, and returns it as-is.

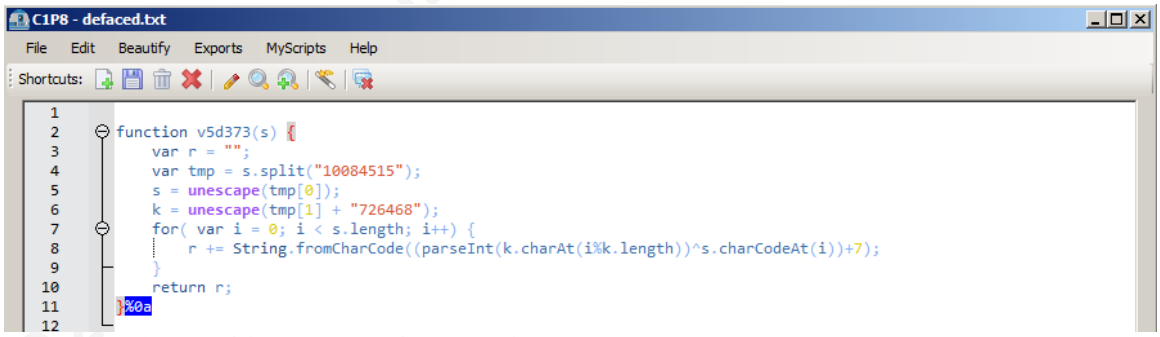

Figure 4 - evaluation of the "Unescape" function with JINT

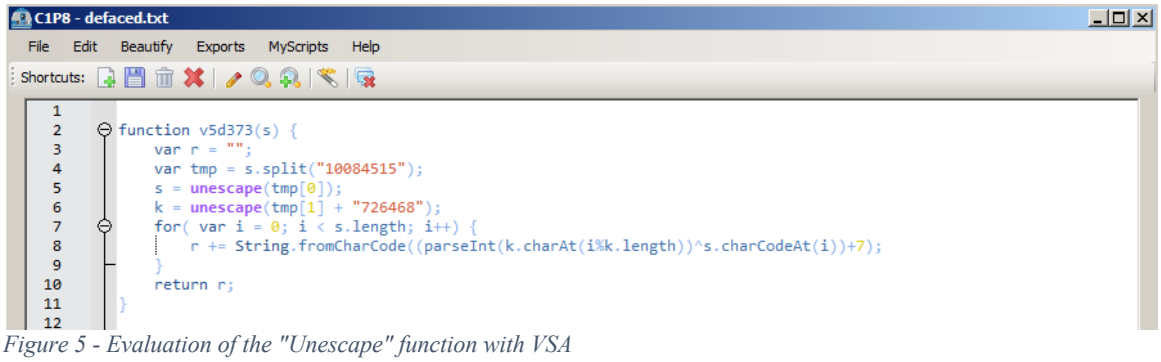

Figure 5 - Evaluation of the "Unescape" function with VSA

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

6.3. ClearScript V8

"V8" is a well-known open-source engine initially developed by "The Chromium Project" for "Google Chrome." "Clearscript V8" is the name of the package, released by Microsoft, which includes V8. The code to implement "ClearScript V8" as interpreter internal to the application is as straightforward as per the previously seen "VSA" and "Jint" engines:

```
private void clearscriptTest(object sender, EventArgs e)
        {
            try
            {
                string gino = scintilla1.SelectedText;
                var v8e = new Microsoft.ClearScript.V8.V8ScriptEngine();
                var alfa = v8e.Evaluate(gino);
                scintilla1.ReplaceSelection(alfa.ToString());
            }
            catch {
                MessageBox.Show("ClearScript error");
            }

        }
```

Table 12 - Code for ClearScript V8

Similar to the other two previously seen "VSA" and "Jint" engines, the implementation shown in Table 12 of "V8" has the same limitations already noted for the other two engines: the first is that it does not include definitions for web browser or PDF viewer specific objects. To overcome this restraint an object definition, similar to the one shown at Table 9 needs to be implemented. The second limitation lies in the fact that this basic implementation is not suited to evaluate an entire source code; nevertheless, it is useful to resolve small code blocks, evaluate math operations within the source code and resolve native JavaScript functions, as seen for "VSA" and "Jint." In the course of preliminary tests carried out for this latter engine, one additional limitation for "V8" was noted: "VSA" and "Jint" successfully resolved an array of integers whose single elements were obfuscated using the "number splitting" obfuscation pattern (mentioned at paragraph 3.2), while "V8" returned an error, and did not resolve the obfuscated array.

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

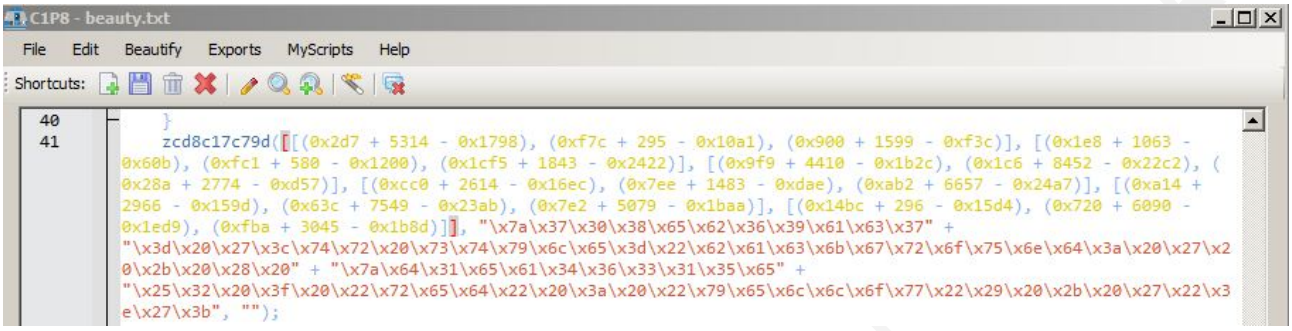

Figure 6 - splitted array elements

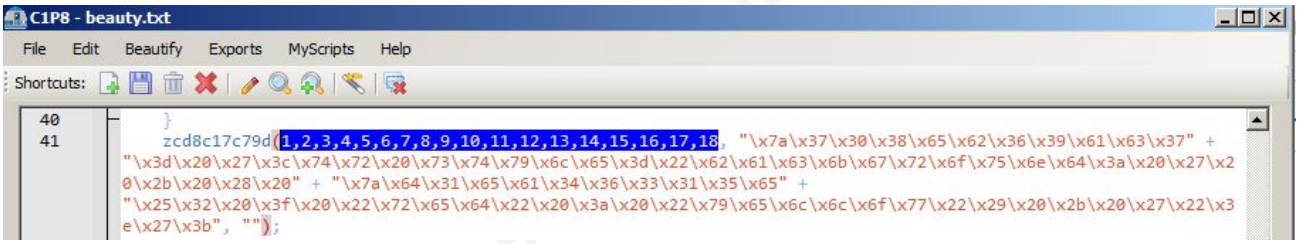

Figure 7 - splitted array elements correctly evaluated by VSA and JINT

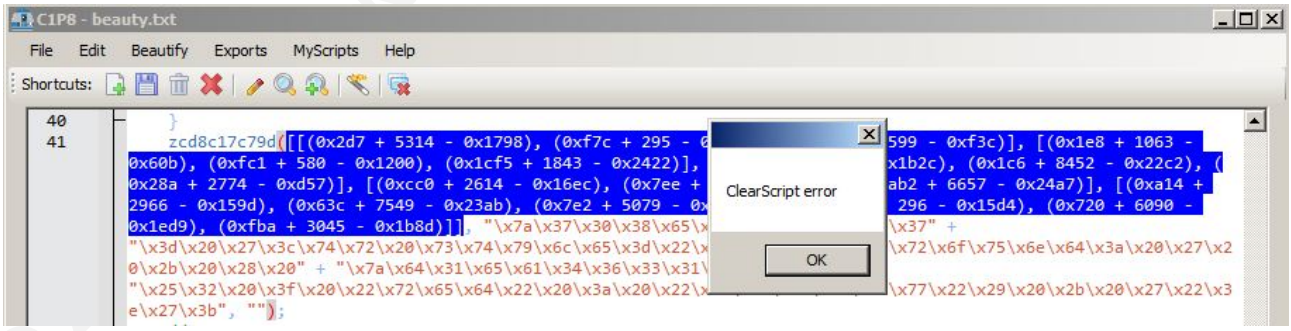

Figure 8 - ClearScript V8 returning a parsing error

6.4. Considerations regarding JavaScript engines

The JavaScript engines tested so far are designed to expand the capabilities of .NET applications (especially server side applications or REPL applications) and are not intended for JavaScript de-obfuscation. As such, a standalone interpreter like SpiderMonkey is surely more versed for this purpose rather the aforesaid engines. Nevertheless, these engines are surely ideal in a context of obfuscating malicious JavaScript code in interactive mode

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

("manual de-obfuscation"), especially in the cases where the obfuscation mechanism is so complex that none of the available tools (including SpiderMonkey and Cscript) can de-obfuscate it with a simple pass. Forensic examiners and malware analysts know that no software is all inclusive and that having multiple tools available in their arsenal is the ideal scenario. For this first reason, the inclusion of the aforesaid JavaScript evaluation engines in a de-obfuscation tool, along with a wrapper for existing standalone interpreters, is a recommended choice. In the next paragraphs (AST de-obfuscation), the "EvalJScript" method shown in Table 8, using the implementation of the VSA engine, will be used again along with functionalities of other packages in order to perform AST evaluation of a simple obfuscation pattern. Implementation of JavaScript engines is then essential if analysts are considering hybrid approaches for de-obfuscation.

7. AST (Abstract Syntax Tree) De-obfuscation

The Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language, where each node of the tree denotes a construct occurring in the source code. The "Esprima" project (Hidayat, n.d.) is one of the most known parsing infrastructures for JavaScript, which also includes AST generation capabilities. In recent years, several researchers started considering AST as a valid method to de-obfuscate malicious JavaScript. The following "string splitting" obfuscation pattern is a theoretical example of how AST can help with code de-obfuscation :

```
var a = "from" + "CharCode";
```

This JavaScript expression translates in the below AST tree, here reported in JSON format:

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

```
{“type”:”Program”,”body”:[{“type”:”VariableDeclaration","declarations":[{"type":"
VariableDeclarator","id":{"type":"Identifier","name":"a"},"init":{"type":"BinaryE
xpression","operator":"+","left":{"type":"Literal","value":"from","raw":"\"from\"
"},"right":{"type":"Literal","value":"CharCode","raw":"\"CharCode\""}}}],"kind":"
var"}],"sourceType":"script"}
```


Table 13 - AST (Json format) for: var a = "from" + "charCode"

After resolving the “string splitting” pattern by concatenating the two strings into one, the expression above transforms as follows:

```
var a = "fromCharCode";
```

The AST equivalent of this latter expression, in JSON format, is as follows:


```
{"type":"Program","body":[{"type":"VariableDeclaration","declarations":[{"type":"
VariableDeclarator","id":{"type":"Identifier","name":"a"},"init":{"type":"Literal
","value":"fromCharCode","raw":"\"fromCharCode\""}}],"kind":"var"}],"sourceType":
"script"}
```


Table 14 - AST (Json format) for: var a = "fromcharCode"

AST de-obfuscation consists of translating the original JavaScript source code into AST, then identifying an obfuscation pattern within the AST Tree, replacing the identified AST pattern with the AST of its de-obfuscated equivalent and finally reverting back the whole AST tree into JavaScript code. At the end of this process, the expected result is obtaining JavaScript code with the identified pattern de-obfuscated.

This de-obfuscation method is much safer than evaluation through JavaScript engines because it can be completely static (as in the above example) or partially static (also referred to as “partial evaluation”); the AST de-obfuscation of the following JavaScript expression “var a = 22 + 4” would in fact imply the addition “22 + 4” being dynamically computed.

One of the first de-obfuscators that uses this technique is free JavaScript tool “Esde-obfuscate.js” (2015), based on Esprima tools and developed by researcher “Igor Null”, also known as “M1el”. In 2015 and 2018, Italian cybersecurity researcher Stefano Di Paola introduced more advanced de-obfuscation concepts and released a new tool named “JStillery”

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

that makes use of AST de-obfuscation techniques. One of the dependencies for this tool, other than Esprima, is another JavaScript free tool named "Escodegen.js", released by Japanese researcher Yusuke SUZUKI in year 2012.This latter script takes an AST as input and returns JavaScript code. According to the tests carried out by the aforementioned researchers who explored AST de-obfuscation first, this newly discovered methodology is a valid way to de-obfuscate code concealed with the "Jsobfu" Metasploit obfuscator.

7.1. AST De-obfuscation in .NET

"Esprima.NET" is the .NET porting for "Esprima", released by developer Sebastien Ros, who also released the Jint JavaScript engine. Obtaining the AST from JavaScript code using Esprima.NET is very simple, and can be achieved using the methods shown below:

```
    private void AnyFunction(object sender, EventArgs e)
        {
            string alpha = scintilla1.SelectedText;
            string beta = ASTParse(alpha);
            scintilla1.ReplaceSelection(beta);
        }

        public static string ASTParse(string source)
        {
           var parser = new JavaScriptParser(source);
            var program = parser.ParseProgram();
            string result = program.ToJsonString();
            return result;
        }
```

Table 15 - Obtaining AST with Esprima.NET

One limitation of the current version of "Esprima.NET" is that it does not have built-in methods to revert from AST to JavaScript code.

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

Given that "Esprima.NET" can return an AST representation in JSON format, a turnaround that was implemented to bypass this limitation was a combination of regular expressions (to identify the AST of the obfuscated pattern that needs to be de-obfuscated) and methods provided by package "Newtonsoft.Json", which is a well-known free library that provides numerous tools for JSON manipulation. For this research, a proof of concept code that performs a simple AST de-obfuscation of a "number splitting" obfuscation pattern was developed. In more details, the proof of concept code in question performs the following actions:

1. Obtains the original source code and translates it into its AST representation;
2. From this AST representation, a JSON pattern (corresponding to a "string splitting" or to a "number splitting" obfuscation pattern) is extracted using regular expressions;
3. The extracted AST pattern is deserialized using Newtonsoft package functionalities;
4. From the deserialized pattern, the exact location of the matching pattern (line and column of either expression start and expression end) is taken, and transformed into "Scintilla control coordinates";
5. The math operation which consists of "number splitting" is evaluated;
6. The result (de-obfuscated equivalent of the number splitting expression) is built;
7. The old, obfuscated JavaScript expression is replaced with its obtained equivalent.

The proof of concept code that will be explained in the next paragraph is a source code that, taking advantage of AST representation and implementing AST partial evaluation concepts, aims to resolve expression "var a = 22 + 4;" into expression "var a = 26;."

7.1.1. Obtaining AST in JSON format with offsets

In order to gain not only the AST representation of source code but also the exact position (line and column) of each instruction, the "ASTParse" method shown in Table 15 is modified by adding some additional options, returning the position for the given expression of

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

the source code. It is important to know the exact location where the expression is contained because once the elements in the AST are evaluated, and a de-obfuscated expression is built, the program must know the exact location of the old code, and replace it with the newly obtained expression.

```
        public static string ASTParse(string source)
        {
        var parser = new JavaScriptParser(source, new ParserOptions { Loc =
true });
       AstJson.Options opt = new
AstJson.Options().WithIncludingLineColumn(true);
       var program = parser.ParseProgram();
       string result = program.ToJsonString(opt);
       }
```

Table 16 - obtaining AST with code line numbers

Once the expression "var alfa = 22 + 4" is translated in AST using the new version of the "ASTParse" method, its JSON representation will contain the start position and end position for each of its elements. The last JSON block, highlighted in red font color at Table 17, is the start and end location of the entire expression. These coordinates will be used to replace the code, after partial evaluation occurs. In this case, the expression "var alfa = 22 + 4;" was found at line 6 starting at column 4 (as there were four spaces before it) and ends at column 22 (since the expression is 18 characters long, including its spaces), as follows:


```
{"type":"VariableDeclaration","declarations":[{"type":"VariableDeclarator","id":{
"type":"Identifier","name":"alfa","loc":{"start":{"line":6,"column":8},"end":{"li
ne":6,"column":12}}},"init":{"type":"BinaryExpression","operator":"+","left":{"ty
pe":"Literal","value":22,"raw":"22","loc":{"start":{"line":6,"column":15},"end":{
"line":6,"column":17}}},"right":{"type":"Literal","value":4,"raw":"4","loc":{"sta
rt":{"line":6,"column":20},"end":{"line":6,"column":21}}},"loc":{"start":{"line":
6,"column":15},"end":{"line":6,"column":21}}},"loc":{"start":{"line":6,"column":8
},"end":{"line":6,"column":21}}}],"kind":"var","loc":{"start":{"line":6,"column":
4},"end":{"line":6,"column":22}}}
```


Table 17 - AST representation for the "var alfa = 22 + 4;" JavaScript expression

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

7.1.2. Regex extracting the JSON pattern

Starting from the AST sample at Table 16, a regular expression that finds this pattern within the entire AST tree is then obtained. Using a regular expression to extract this pattern is a convenient turnaround in order to first identify whether this pattern exists in the source code to de-obfuscate, and then in order to obtain a JSON block that has a known structure, and is therefore easy to deserialize:

```
string regespress1 =
"{\"type\":\"VariableDeclaration\",\"declarations\":\[{\"type\":\"VariableDeclara
tor\",\"id\":{\"type\":\"Identifier\",\"name\":\"[a-zA-Z0-
9_]+\",\"loc\":{\"start\":{\"line\":[0-9]+,\"column\":[0-
9]+},\"end\":{\"line\":[0-9]+,\"column\":[0-
9]+}}},\"init\":{\"type\":\"BinaryExpression\",\"operator\":\"[+-
/*%^]\",\"left\":{\"type\":\"Literal\",\"value\":[0-9]+,\"raw\":\"[0-
9]+\",\"loc\":{\"start\":{\"line\":[0-9]+,\"column\":[0-
9]+},\"end\":{\"line\":[0-9]+,\"column\":[0-
9]+}}},\"right\":{\"type\":\"Literal\",\"value\":[0-9]+,\"raw\":\"[0-
9]+\",\"loc\":{\"start\":{\"line\":[0-9]+,\"column\":[0-
9]+},\"end\":{\"line\":[0-9]+,\"column\":[0-
9]+}}},\"loc\":{\"start\":{\"line\":[0-9]+,\"column\":[0-
9]+},\"end\":{\"line\":[0-9]+,\"column\":[0-
9]+}}},\"loc\":{\"start\":{\"line\":[0-9]+,\"column\":[0-
9]+},\"end\":{\"line\":[0-9]+,\"column\":[0-
9]+}}}\],\"kind\":\"var\",\"loc\":{\"start\":{\"line\":[0-9]+,\"column\":[0-
9]+},\"end\":{\"line\":[0-9]+,\"column\":[0-9]+}}}";
```

Table 18 - Regular expression to extract JSON pattern

7.1.3. Deserializing the JSON pattern

In case one or more JSON patterns representing an obfuscated JavaScript expression are identified via the regular expression, the patterns themselves can be temporarily saved into a string variable for their deserialization. The “Newtonsoft JSON” package, available via the Nuget repository, is a de facto standard for JSON manipulation and a .NET package version is also available. Before writing the deserialization function, the JSON pattern needs to be defined in a dedicated struct, like the one shown in the table below:

Author Name, email@address

© 2021 The SANS Institute Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

```
            public struct ASTstruct1
            {
                public string Tipo;
                public string nomevariabile;
                public string operatore;
                public int destra;
                public int sinistra;
                public int offsetinizioriga;
                public int offsetiniziocolonna;
                public int offsetfineriga;
                public int offsetfinecolonna;
    }
```

Table 19 - Struct to define a JSON pattern

One of the advantages offered by Newtonsoft is that, for this specific proof of concept, it is not necessary to define the entire structure of the record but rather only requires a structure containing only the elements needed for this deserialization. In particular, in the above shown struct, only the following elements are defined:

1. The “var” type that is assigned to the variable (struct member “Tipo”);
2. The name of the variable (“alfa”, stored in struct member “nomevariabile”);
3. The operator type (“+”, stored in struct member “operator”);
4. The operand at the right (“22”, stored in struct member “destra”);
5. The operand at the left (“4”, stored in struct member “sinistra”);
6. The row number of the beginning of the expression (struct member “offsetinizioriga”);
7. The column number of the beginning of the expression (member: “offsetiniziocolonna”);
8. The row number of the end of the expression (member: “offsetfineriga”);
9. The column number of the end of the expression (member: “offsetfinecolonna”);

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

Once the struct is defined, a dedicated function (named “assegna1”, in the following code example) will take the JSON pattern in question (“estratto”) as argument and return a structure of type “ASTstruct1”. The type of the “estratto” pattern that we are passing to the function as argument has type “dynamic”, which is a new type introduced from C# version 4:

```
public ASTstruct1 assegna1(dynamic estratto) {
       ASTstruct1 aststruct1 = new ASTstruct1();

       aststruct1.Tipo = estratto.kind;
       aststruct1.nomevariabile = estratto.declarations[0].id.name;
       aststruct1.operatore = estratto.declarations[0].init.operator;
       aststruct1.destra = estratto.declarations[0].init.right.value;
       aststruct1.sinistra = estratto.declarations[0].init.left.value;
       aststruct1.offsetinizioriga = estratto.loc.start.line;
       aststruct1.offsetiniziocolonna = estratto.loc.start.column;
       aststruct1.offsetfineriga = estratto.loc.end.line;
       aststruct1.offsetfinecolonna = estratto.loc.end.column;
       return aststruct1;
         }
```

Table 20 - function extracting values from the JSON pattern and assigning them to the members of the struct

7.1.4. Obtaining the offset for the original expression

Esprima.NET is able to generate an AST representation in JSON format that also contains the start and end location for each of its elements. In particular, the beginning and the end of an expression is defined with four integers: starting line, starting row, ending line and ending row. The Scintilla control defines the start and end of a given portion of text by using two integers which are absolute offsets (character number) within the control itself. A method that operates a conversion between the two different formats is represented in the below code sample:

```
  public int getposition(int riga, int colonna) {
    int coordinata;
    ScintillaNET.Line linea = new ScintillaNET.Line(scintilla1, riga - 1);
```


© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

```
    coordinata = linea.Position + colonna;
    return coordinata;
       }
```

Table 21 - method converting JSON offsets into Scintilla offsets

7.1.5. Defining the main code

After structs and necessary functions are defined, the main code that performs AST review can be developed, which conducts a pattern recognition via regex and then performs a simple partial evaluation and replacement of the original code ("number splitting" pattern) with its de-obfuscated equivalent. The code block below makes use of the "EvalJScript" method, described at paragraph 6.1 (Table 8) to evaluate the math expression, and makes use of the "ReplaceSelection" Scintilla API, described at paragraph 4 (Table 1) to replace the obfuscated expression with the de-obfuscated one. In more detail, the code below executes the following:

1. Compiles the regular expression shown in Table 17;
2. Checks for all eventual matches and, if there is one or more match, deserializes it with the "JsonConvert.DeserializeObject" Newtonsoft method;
3. The "Assegna1" functiontakes the values needed for this proof of concepts and assigns each of them to the corresponding struct members;
4. Destination offsets within Scintilla are determined with function "getposition";
5. The "EvalJScript" performs evaluation of the code;
6. The "SetSelection" Scintilla method programmatically selects the expression to be replaced;
7. The "ReplaceSelection" Scintilla method replaces old code with new code.

```
Regex regex = new Regex(regespress1);
MatchCollection match = regex.Matches(romolo);
   if (match.Count >= 1)  {
        foreach (Match elem in match) {
             try  {
```


© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

```
                dynamic selectedJSON =
JsonConvert.DeserializeObject(elem.Value);
                ASTstruct1 patn = assegna1(selectedJSON);
                int iniziolinea = getposition(patn.offsetinizioriga,
patn.offsetiniziocolonna);
                int finelinea = getposition(patn.offsetfineriga,
patn.offsetfinecolonna);
                valutata = EvalJScript(patn.destra + patn.operatore +
patn.sinistra).ToString();
                finale = patn.Tipo + " " + patn.nomevariabile + " = " +
valutata + ";";
                scintilla1.SetSelection(iniziolinea, finelinea);
                scintilla1.ReplaceSelection(finale);
                      }
                      catch { }
                   }
             }
             else {
                 // Any action
             }
```

Table 22 - main function for the AST de-obfuscation routine

The test of the above code implemented within the application was successful, having transformed the expression “var alfa = 22 + 4;” into expression “var alfa = 26;”, Figure 9 shows the originally obfuscated expression and Figure 10 shows the result of the method defined throughout section 7:

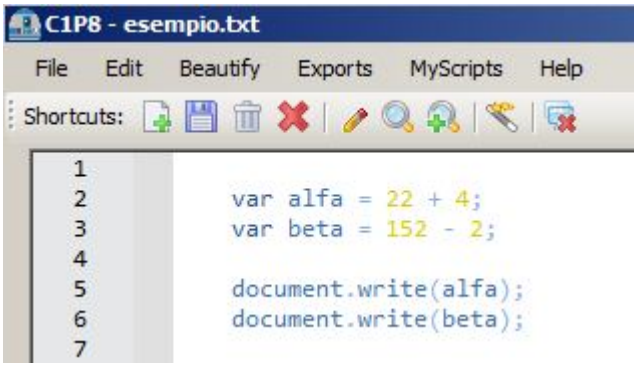

Figure 9 - Code before AST evaluation

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

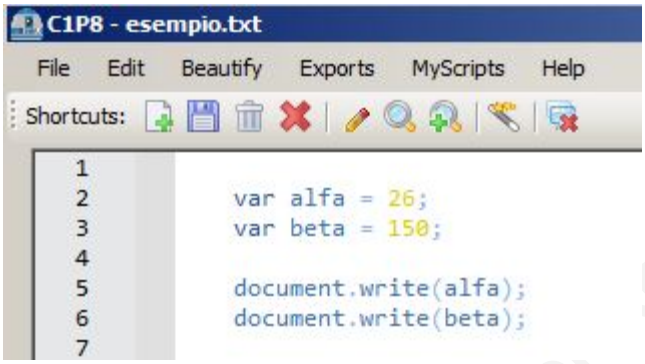

Figure 10 - Code after AST evaluation

8. Finalizing a JavaScript De-obfuscation Tool

In order to test the validity and efficiency of all the explained C# code, an application named "C1P8" was developed and compiled specifically for this research paper. The compiled version of this application is available in a dedicated GitHub repository, at address https://github.com/c1p8-RN/C1P8.

9. Testing the Application Against Malicious JavaScript

As a final test to assess the efficiency of the above code, the obtained application was tested against malicious code selected from a public GitHub repository for malware samples, already cited in the "Research Methodology" section of this paper. The scripts that were chosen to test the application developed for this research are "2015-07-17-malicious-JavaScript-from-stepanovichon.com.js" (available in the "misc" subfolder of said repository) and "jquery_injected.js" (Angler Exploit Kit dropper), available in the "angler/12" subfolder. Both malicious scripts were selected because, in the same locations of the repository, the de-obfuscated versions of both JavaScript codes are present.The availability of the solution for the de-obfuscation helps with checking the correctness of the obtained results. The outcome of the final test for the above .NET code snippets, and the application, is described in the following sections.

Author Name, email@address

© 2021 The SANS Institute, Author retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

9.1. Malicious code “2015-07-17-malicious-JavaScript-from-stepanovichon.com.js”

The obfuscated code for this malware sample consists in one “Unescape()” native JavaScript function, containing obfuscated code in the form of escaped characters. This is only the first obfuscation layer for this script, which is easily obfuscated with either the “EvalJScript” method (Table 8), or with the “Unescape” .NET native method The obtained code is another obfuscated JavaScript source code (second obfuscation layer). SpiderMonkey and Cscript both failed to evaluate this second obfuscation layer, therefore a manual de-obfuscation method was used. The de-obfuscated code shown in Figure 11 was obtained by using the below described functionalities:

1) Function names were renamed with the “FindReplace” method (Table 2) in order to assign more meaningful and readable names to the key functions of the malicious code;
2) The variables indicated in the Figure 11 with number “2” originally were strings encoded with escape characters. Those strings were manually decoded with the “Unescape” method described at paragraph 2;
3) Several splitted strings were concatenated manually: Scintilla.NET is a library that offers text editing functionalities, allowing source code to be manually tweaked during the de-obfuscation process without the use of additional external tools.

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

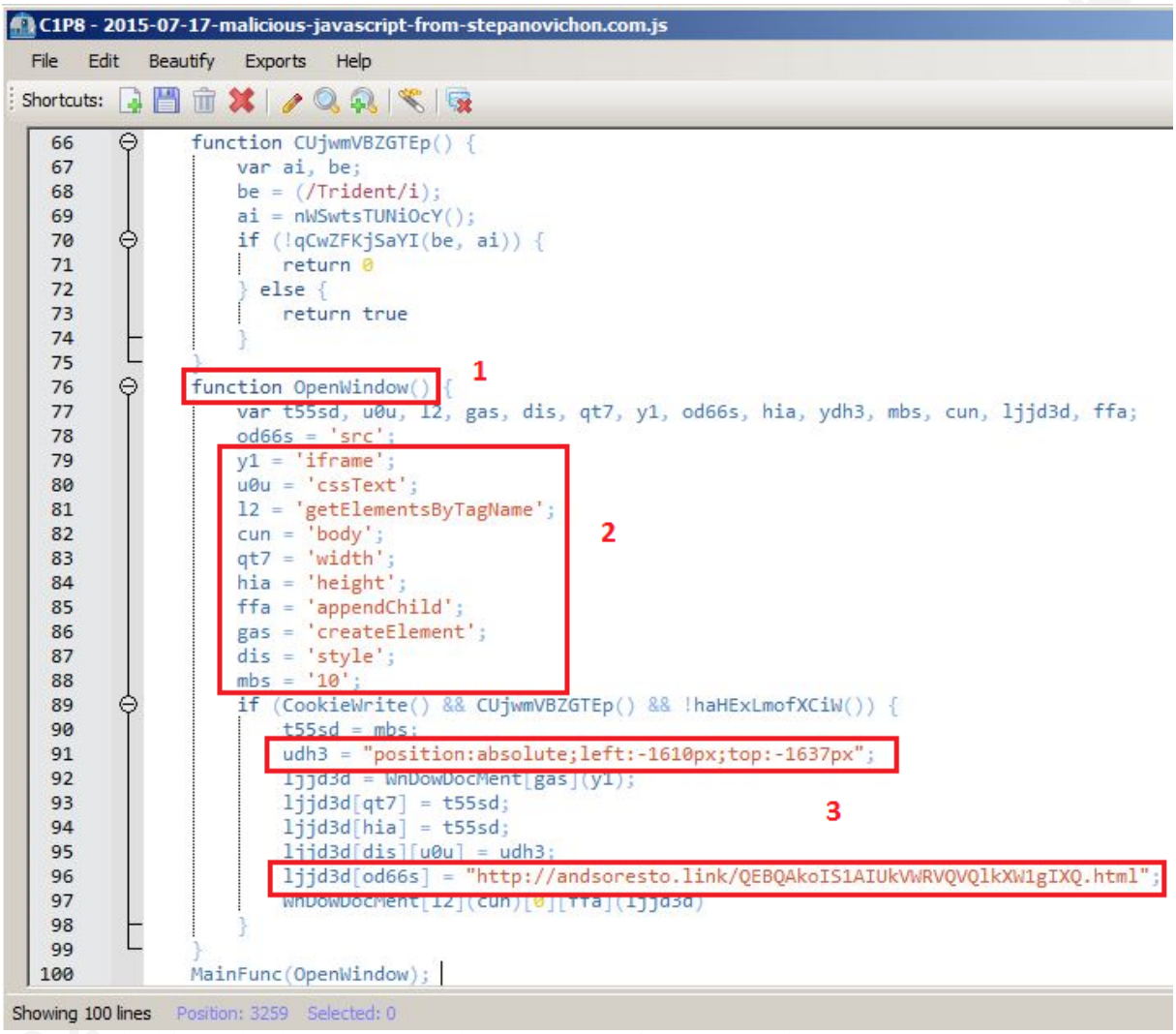

Figure 11 - de-obfuscated code "2015-07-17-malicious-JavaScript-from-stepanovichon.com.js"

9.2. Malicious code "jquery_injected.js"

The obfuscated code for this second sample consists of one simple decoding routine, which uses native JavaScript functions "fromCharCode()", "parseInt()" and "substring()" within a "for" loop, and two nested "eval()" functions to run said code after being decoded. Once the "eval()" functions are manually replaced with "document.write()" to disarm the script, the code was processed with the "SpiderMonkey" embedded functionality, since all the other evaluation engines failed to evaluate the obtained code. The result was a second

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

obfuscation layer, consisting in sequence of decimal numbers that are decoded through a "fromCharCode" native function:

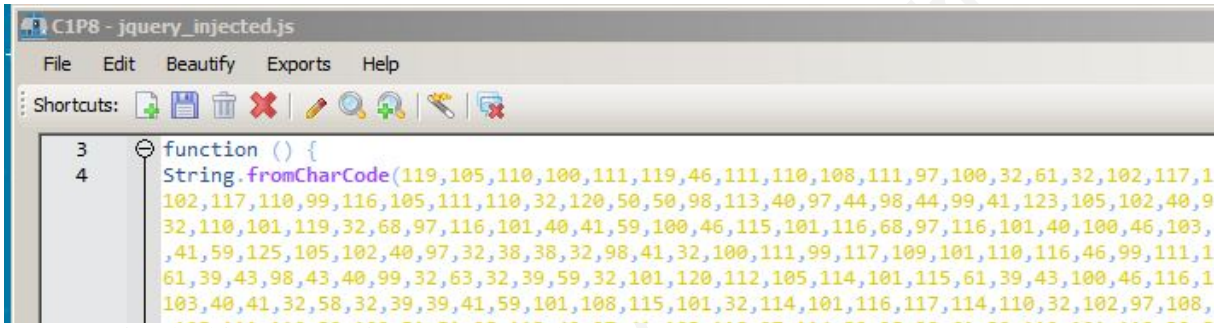

Figure 12 - first de-obfuscation pass for "Jquery_Injected.js"

This native function is successfully processed with both the "Jint" or "VSA" engines, since these two engines are ideal options when resolving native encoding and decoding JavaScript functions.

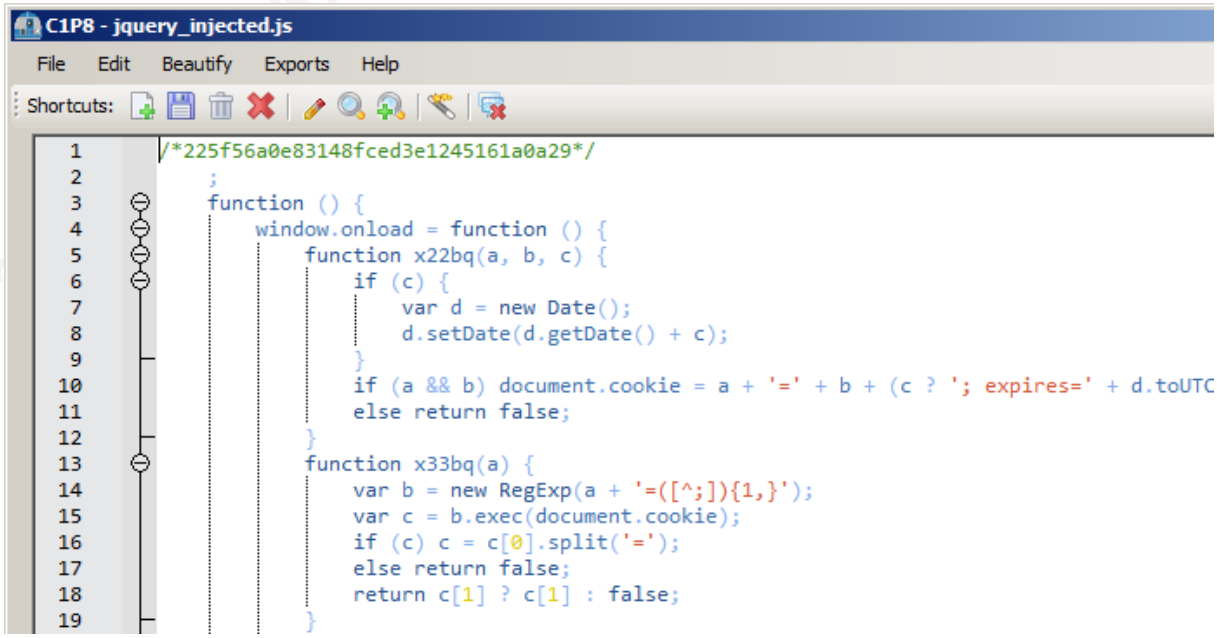

Figure 13 - final de-obfuscated code for "Jquery_Injected.js"

https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

10. Conclusion

For the purpose of de-obfuscating malicious JavaScript, the JavaScript language itself and the cited software developed in JavaScript (Esprima, Escodegen and all the available tools based on these infrastructures) represent a more mature avenue for the malicious JavaScript reverse engineering. In particular, the "Esprima.NET" project, which does not currently have the capability of reversing from AST into source code, has the potential of becoming a new "must have" package even for JavaScript De-obfuscation. Nevertheless, the current "Esprima.NET" package version offers an efficient and easy to use AST translation which opens AST de-obfuscation paths for the .NET world.

The outcome of the tests carried out and described at section 9 stresses the importance of having multiple tools available in the arsenal of an analyst: de-obfuscation of the malicious code was possible by using different techniques and different .NET API. SpiderMonkey, used with the "object.js" script, still remains one of the best tools for code evaluation. Despite this, this standalone interpreter failed to evaluate more complex code, making manual de-obfuscation necessary. One key for successful de-obfuscation is the availability of the most numerous techniques and tools possible.

"AST partial evaluation" is a new avenue for the JavaScript de-obfuscation world. Developed in 2015, it is a still young technique but it has a great potential. AST partial evaluation cannot resolve every type of obfuscated code but it represents a safer way of de-obfuscation since it is completely static or only partially dynamic which presents a lesser risks of autopwnage.

The present research was completed with the development of a de-obfuscation tool, named "C1P8", which has been made available in its first beta version in GitHub, with the purpose of concretely demonstrating the validity of the shown C# code. It aims at receiving inputs and contributions from the malware analysis community and from the programming community.

Author Name, email@address

© 2020 The SANS Institute https://t.me/learningnets Author retains full rights.

References

Krill, Paul, *JavaScript creator ponders past, future*. (2008, June 23). Infoworld. https://www.infoworld.com/article/2653798/JavaScript-creator-ponders-past--future.html

(n.d.), *Geeks on Security*. (2020, September 11). GitHub. https://github.com/geeksonsecurity/js-malicious-dataset

Pardo Lopez, Alejandro, *Seek And De-obfuscate - Uncovering JavaScript* (2008). Master's Thesis no 587, Master in Computer Science . Radboud University Nijmegen. http://www.cs.ru.nl/mtl/scripties/2007/AlejandroLopezScriptie.pdf

Xu, F., Zhang, F., Zhu, S., *The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study*. (2012). Department of Computer Science and Engineering, Pennsylvania State University. http://www.cse.psu.edu/~sxz16/papers/malware.pdf

(Hodgson, Paul),*Scintilla and SciTE*. (2020, September 11). http://www.scintilla.org

Slusser, Jacob, *Scintilla.NET.* (2018, October 31). GitHub. https://github.com/jacobslusser/ScintillaNET

Towner, Steve ("Stumpii"), *ScintillaNET-FindReplaceDialog.* (2017, July 6). GitHub. https://github.com/Stumpii/ScintillaNET-FindReplaceDialog

SpiderMonkey: The Mozilla JavaScript runtime. (n.d.). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

Author Name, email@address

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

Strahl, Rick, *Evaluating JavaScript code from C#.* Rick Strahl's Web Log. (2007, February 14) https://weblog.west-wind.com/posts/2007/feb/14/evaluating-JavaScript-code-from-c

Zeltser, Lenny, *Remnux "Objects.js"*. (n.d.) https://github.com/REMnux/salt-states/blob/master/remnux/config/objects/objects.js

MDN Contributors, *Strict mode*. (2020, September 18). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Ros, Sebastien, *Sebastienros/jint.* (2013). GitHub. https://github.com/sebastienros/jint

Ariya, Hidayat, *Esprima.* (n.d.), https://esprima.org

"M1el ", *M1el/esde-obfuscate.* (2015). GitHub. https://github.com/m1el/esde-obfuscate

Di Paola, Stefano, *Advanced JS De-obfuscation via AST and partial evaluation.* (2015, October 28). Minded Security Blog. https://blog.mindedsecurity.com/2015/10/advanced-js-de-obfuscation-via-ast-and.html

Di Paola, Stefano, *Mindedsecurity/JStillery.* (2017). GitHub. https://github.com/mindedsecurity/JStillery

Ros, Sebastien, *Sebastienros/esprima-dotnet.* (n.d.). GitHub. https://github.com/sebastienros/esprima-dotnet

Author Name, email@address

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.

Newtonsoft JSON, *Newtonsoft JSON.* (n.d.).Nuget Package Repository for .NET. https://www.newtonsoft.com/json

Nardella, Roberto, *C1P8.* (November 2020). GitHub https://www.github.com/C1P8-RN/C1P8

Author Name, email@address

© 2021 The SANS Institute, Author Retains Full Rights
https://t.me/learningnets
© 2020 The SANS Institute Author retains full rights.