Advanced Cyberchef Techniques - Defeating Nanocore Obfuscation With Math and Flow Control

embeeresearch.io/advanced-cyberchef-techniques-defeating-nanocore-obfuscation-with-math-and-flow-control/

Matthew September 3, 2024

CyberChef Tutorials

Applying Flow Control and Mathematical operators to deobfuscate a .vbs loader for Nanocore malware.

Introduction

Cyberchef is an incredible tool with powerful features that are rarely documented and can significantly aid an analyst in their efforts to deobfuscate malware.

Today we will be investigating such features and how they apply to defeating the obfuscation of a recent `.vbs` loader for Nanocore malware.

Our Analysis and Deobfuscation Will Cover...

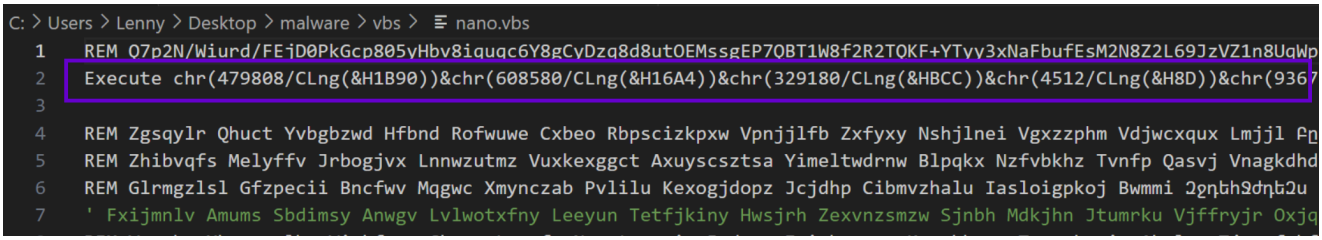
- ASCII Charcodes and Character Conversions
- Alternating Decimal and Hex Values
- Alternating Mathematical Operations (Addition/Division)
- Flow Control and Isolation of Values Using Subsections.
- Lots of regex!

SHA256:`c6092b1788722f82280d3dca79784556df6b8203f4d8f271c327582dd9dcf6e1`

Initial Analysis and Overview of Obfuscation

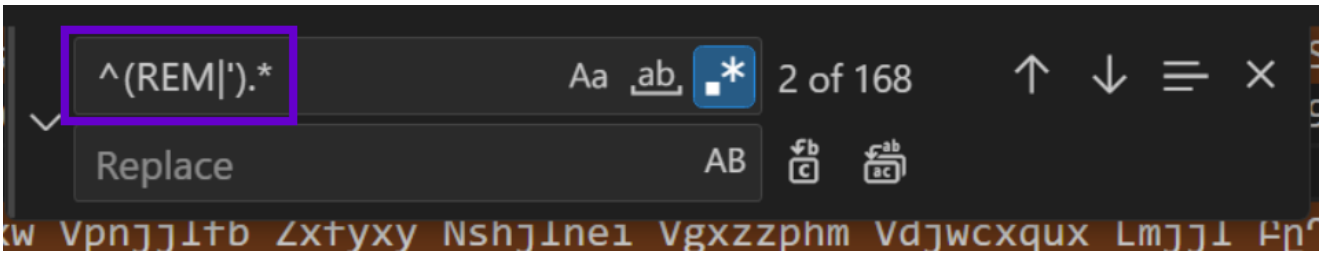
The sample in it's initial state contains ~160 lines of code. The majority of this consists of comments that don't contribute to the functionality of the code.

The primary piece of code exists on line 2 and can be seen below. Our analysis will focus only on this line of code.

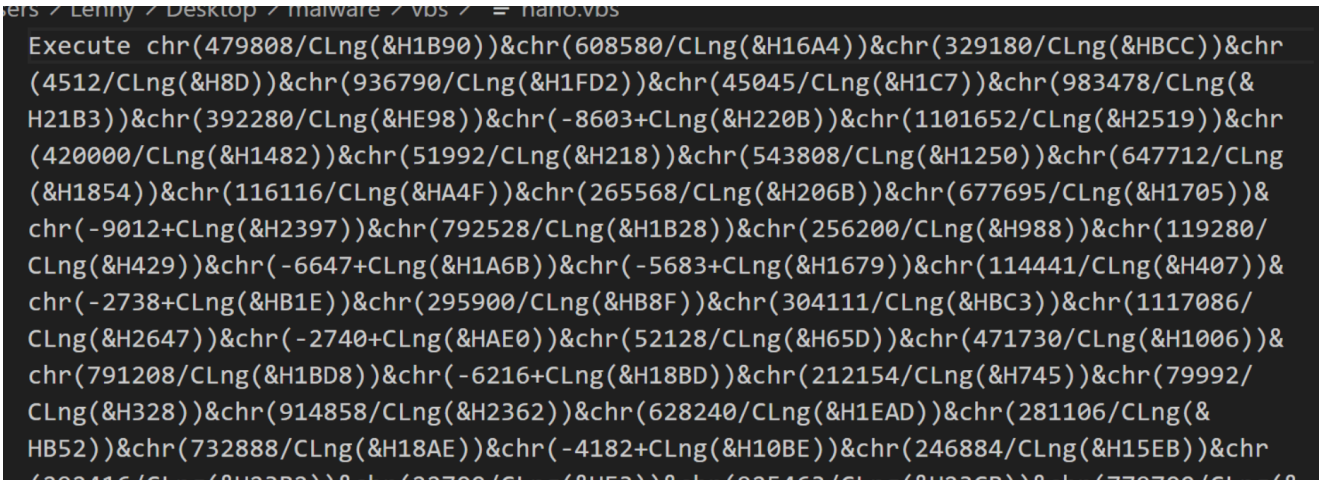


Since our focus is going to be on line 2, we can ignore the remainder of the initial script and remove them using a regular expression.

The goal of the regular expression is identify lines that begin with `REM` or `'`, and to capture everything on that line that follows `.*`



Executing the regular expression as a Find/Replace results in the following content. The comments are now removed and we can focus only on line 2 and it's obfuscation tactics.

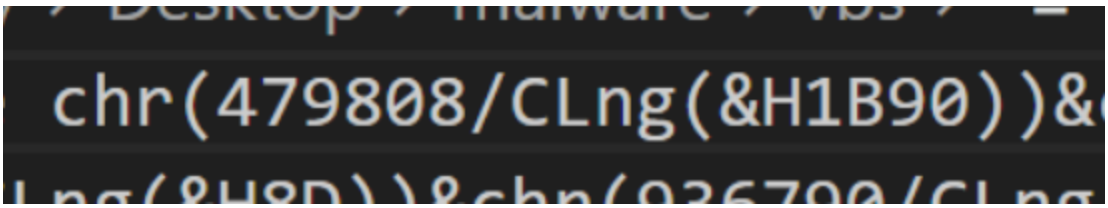


Intial Review of Obfuscation

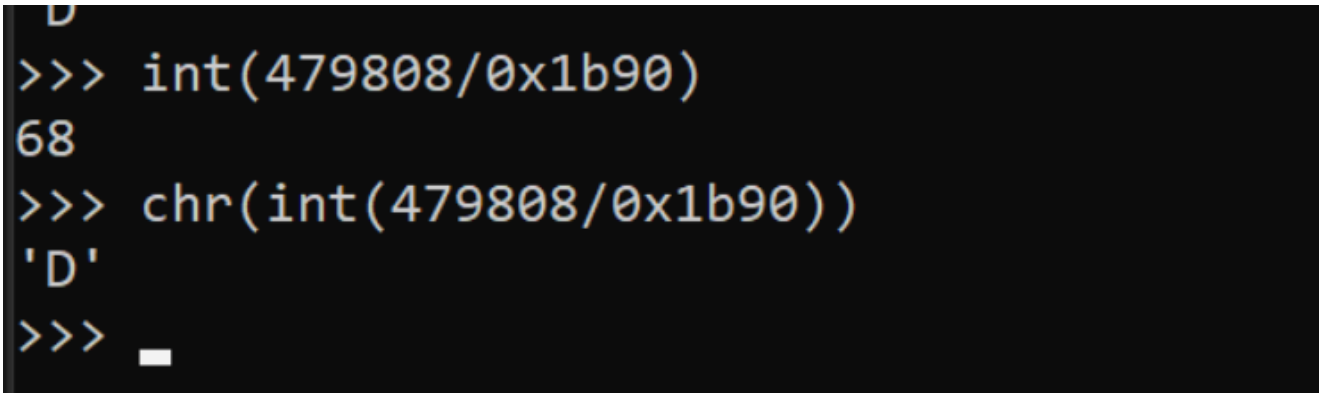
The obfuscation consists of the same pattern repeated over and over again to produce single characters. These characters are concatenated together to form the deobfuscated code.

There are 3 primary pieces of the obfuscation.

- `479808` - Large Decimal Value, this will be converted into a smaller number using math operations.
- `(&H1b90)` - This is a vbs representation of the hex value `0x1B90`.
- `CLng` - This is the function "Change Long", this converts the hex representation into a numerical value.
- `/` - This divides the numbers 479808 and 0x1b90. Resulting in a value in the ASCII range.
- `chr` - The result of the division is converted into a character which will form part of the resulting script.



The logic is more clear when shown in Python. Here we can see that `chr(479808/Clng(&H1B90))` is equal to the character `D`.



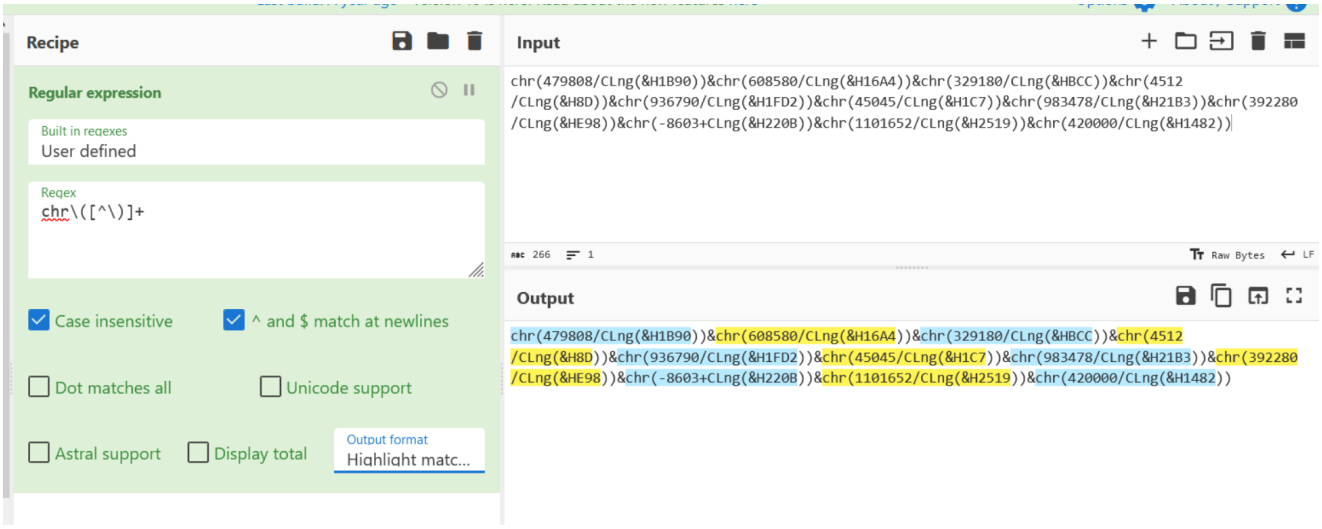
We've now identified the core concept of the obfuscation, so we can go ahead and recreate this in Cyberchef for the entire obfuscated content.

Deobfuscation With Cyberchef

The obfuscation has now been identified, so we can begin to recreate the logic in Cyberchef.

We can begin by isolating the encoded portions with a regular expression `chr\([^\)]+`.

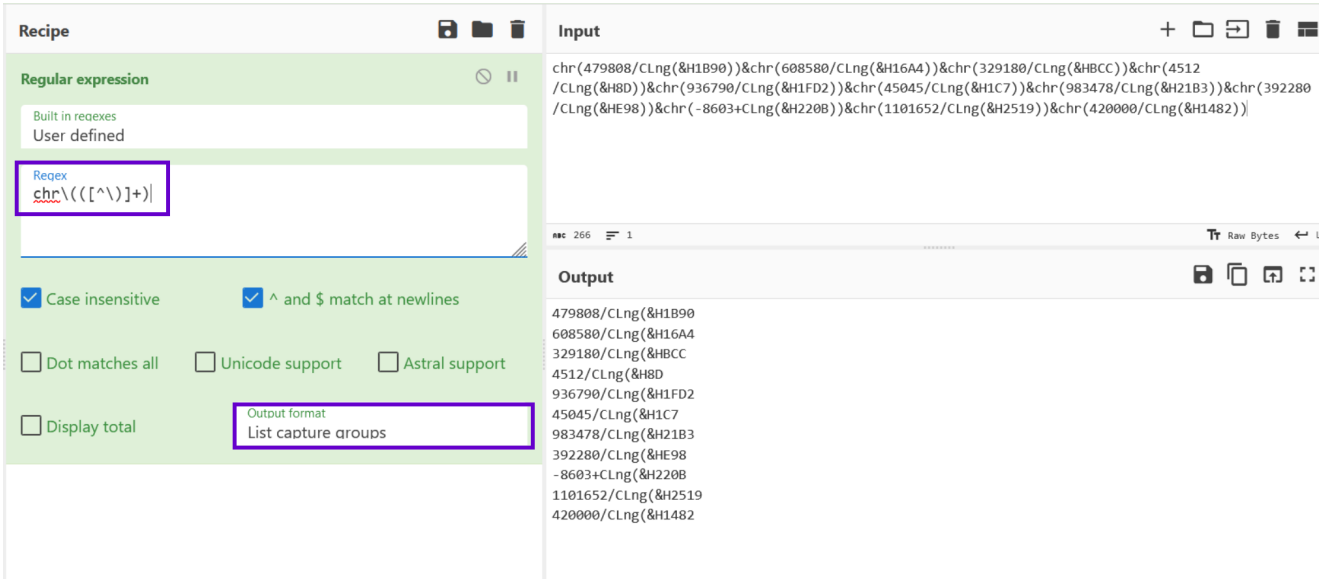
> For the sake of prototyping, we have selected only a small portion of the obfuscated code. This will allow us to get the recipe working before adding the complete script at the end of our analysis.



Isolating Values With Regular Expressions and Capture Groups

Once the regex is matching as intended using "Highlight Matches", we can change to "List Capture Groups".

This will list out the encoded portions on their own individual lines.

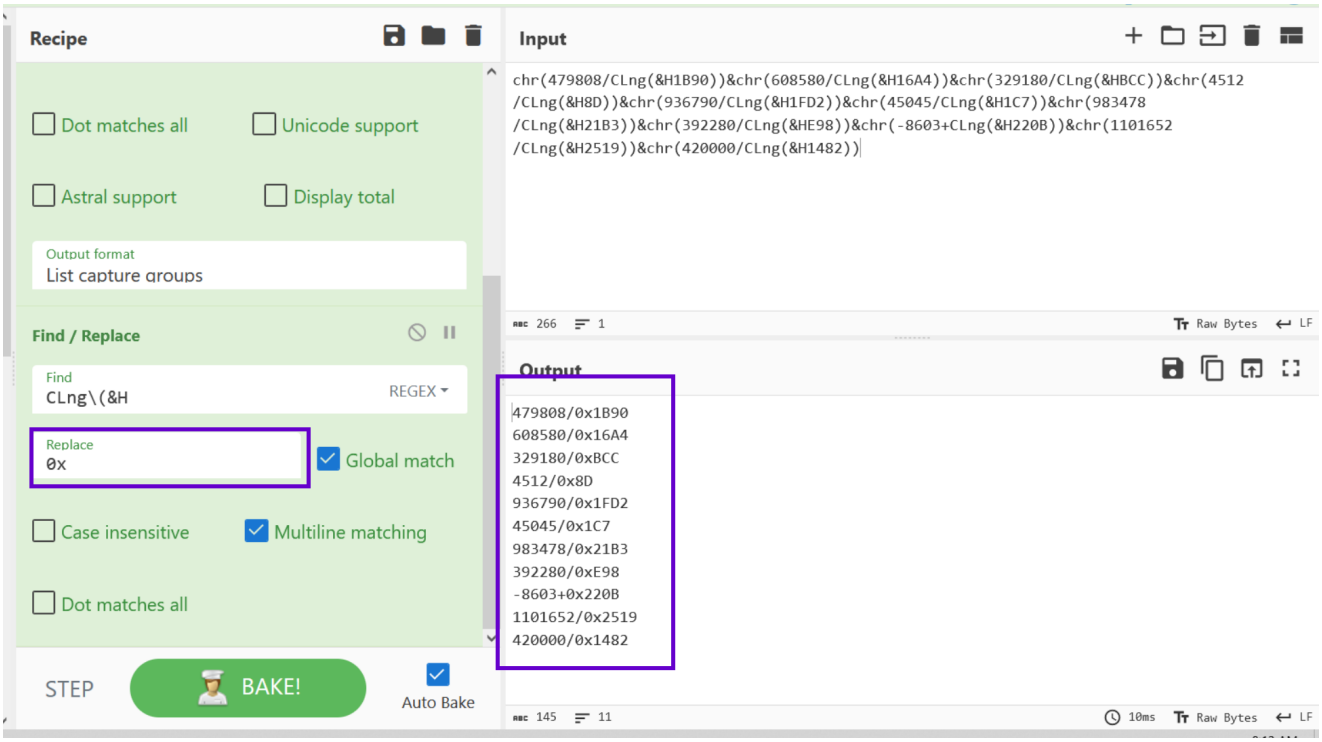


Normalising Hexadecimal Content

We now want to clean up the second half of each line by removing the references to `CLng(&H`.

The original code is in a format that Visual Basic understands. We want to be in a format that can be understood by Cyberchef. Our primary goal is to make sure that Cyberchef knows the difference between the hex and decimal numbers.

We can do this with a Find/Replace operation, which will replace the `CLng(&H` with a `0x` .



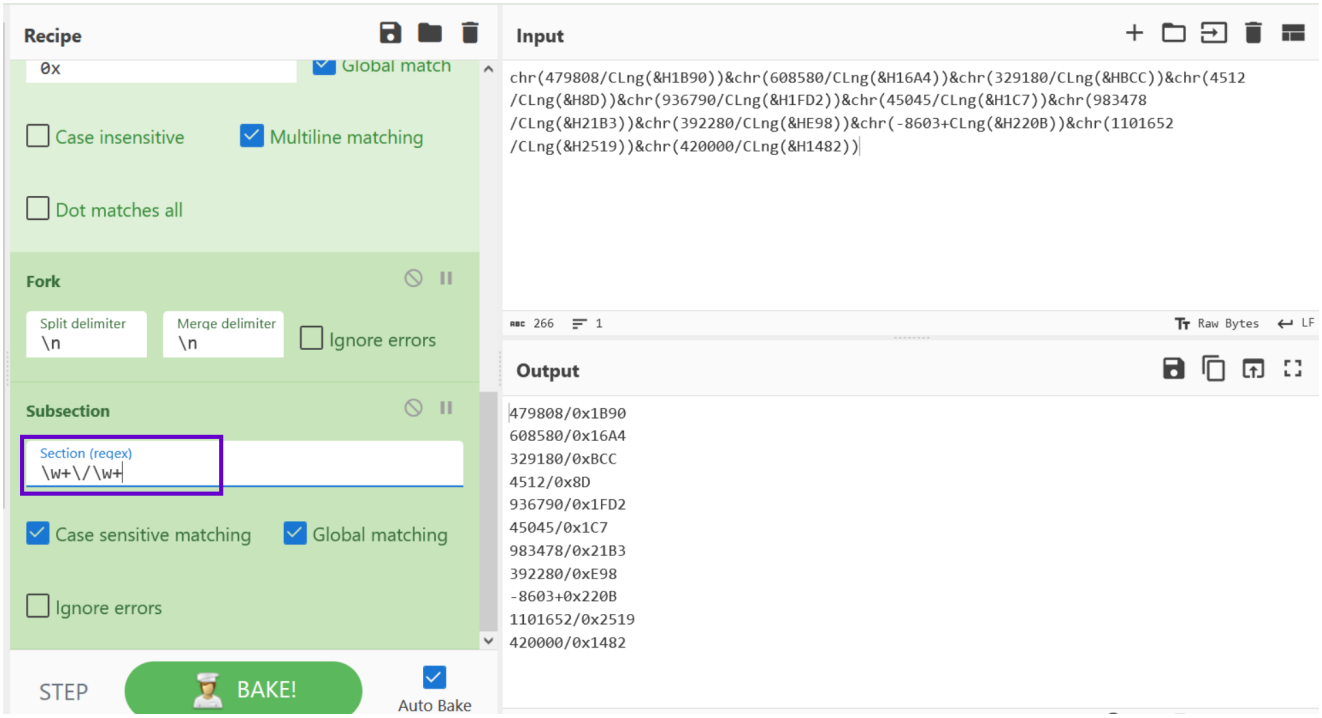
Here is where things start getting more complicated....

As we saw before, the decimal and hex values are separated by mathematical operators. The operators are mostly division `/` but occasionally are addition `+` as well.

If we apply a division operator, it will break the lines that require addition. And vice versa. This means we need to separate the lines of code that require different mathematical operators.

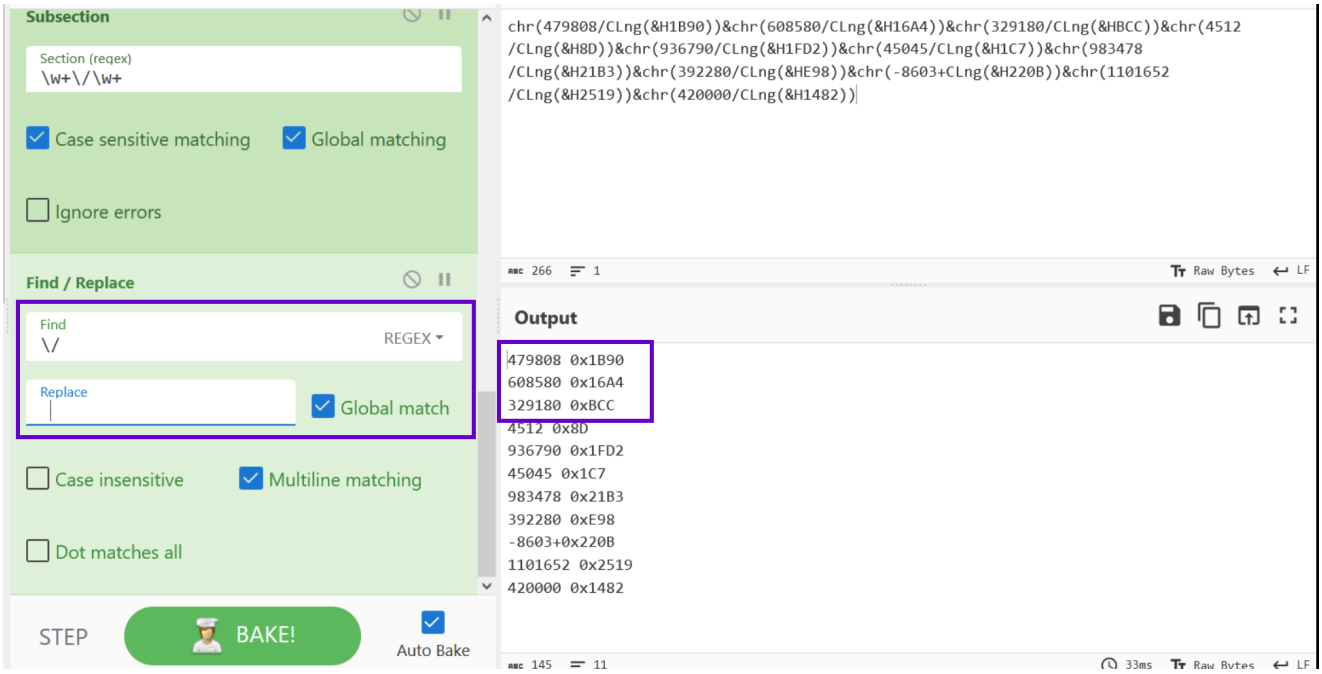
We can do this with Regular Expressions and a `Subsection` operation. A subsection will apply future operations only to lines that match the provided regex.

Below we can see the regular expression of `\w+\/\w+`, this will isolate the lines of code that contain a division operator.



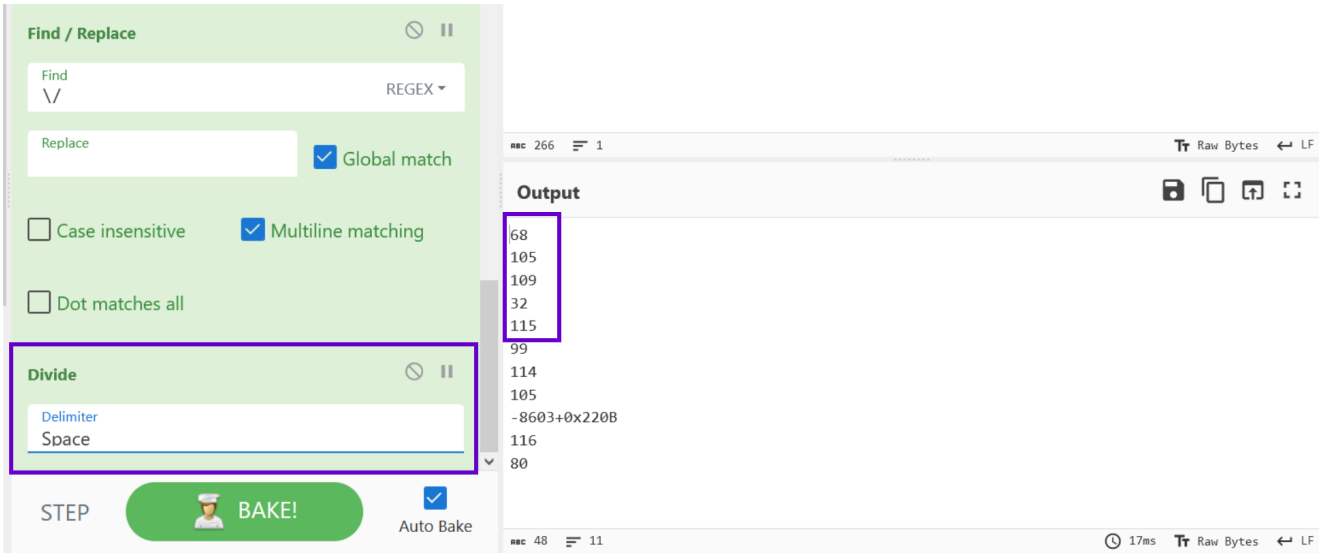
Before applying a division operation, we need to add a delimiter to our divided values. Most math operations in Cyberchef require a "list" of values rather than an equation.

The TLDR here is that we need to turn the `/` into spaces. Luckily we can do this with a simple `Find/Replace`.



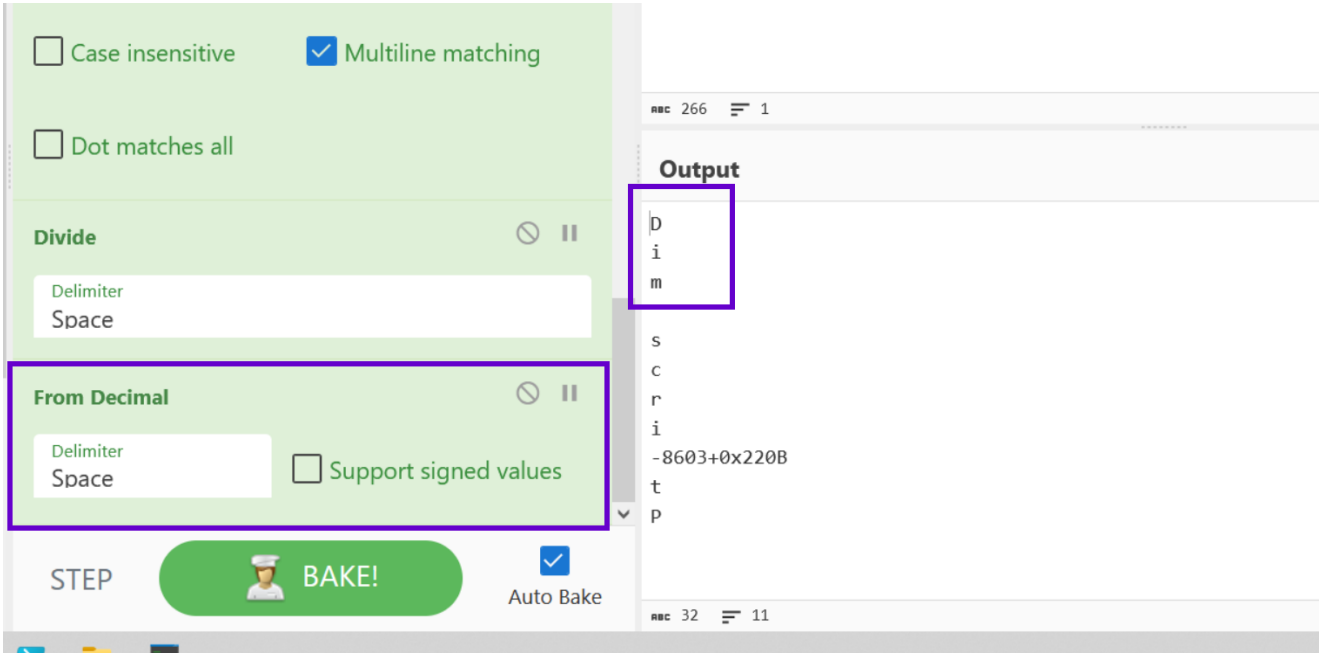
Now that we have applied spacing on the division lines, we can apply a Divide operation and specify a space delimiter.

We can see that this converts the division lines into their repective ASCII charcodes.



With the Charcodes ready, we can apply a simple "From Decimal" to produce the relevant ASCII character.

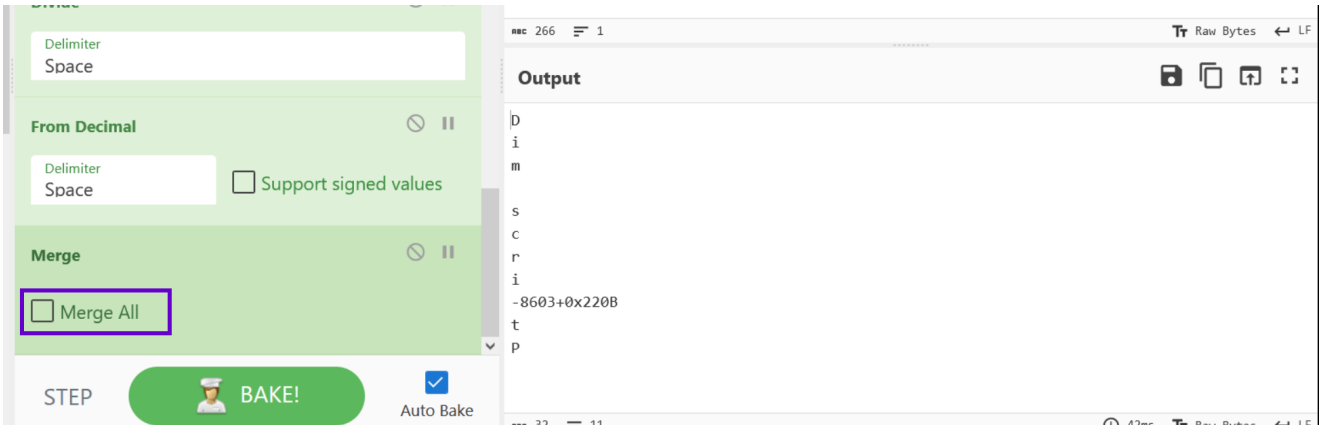
Now we can see the beginning of the decoded script.



Now we need to deal with the lines of code containing addition `+` operators.

Since we previously applied a subsection, we need to leave the subsection and change it to focus on the addition lines.

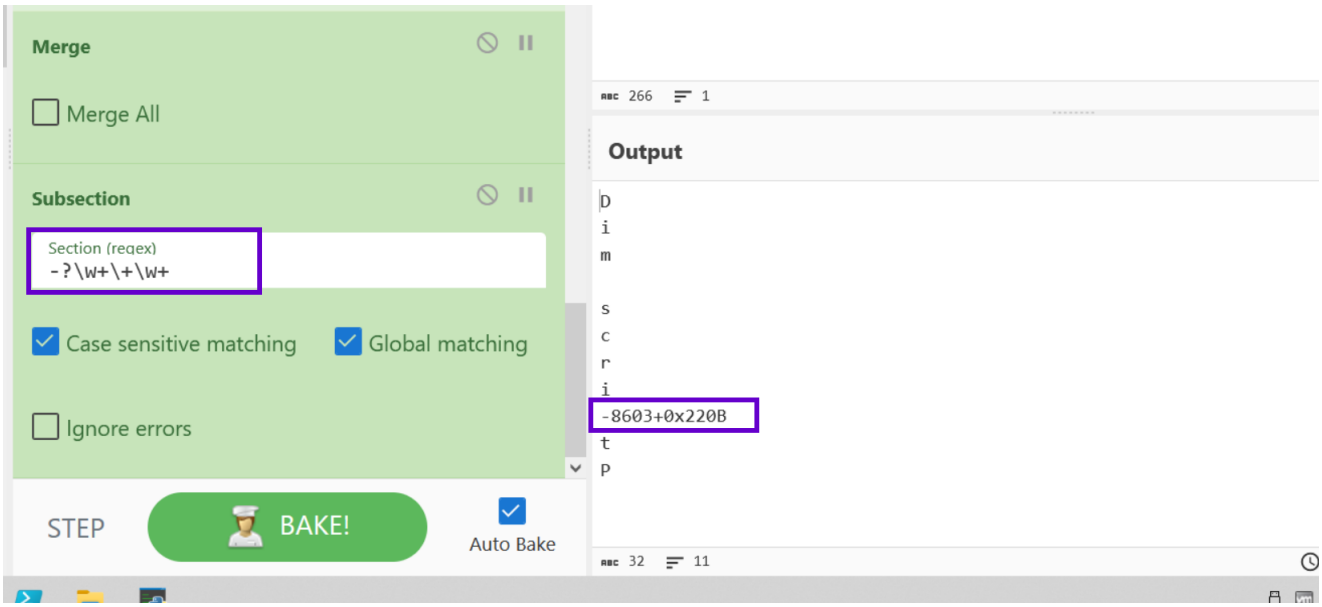
To leave a subsection, we can apply a `Merge` operation. We should also uncheck "Merge All" as there is only a single subsection that we want to leave.



Subsections and Isolating Specific Lines of Content

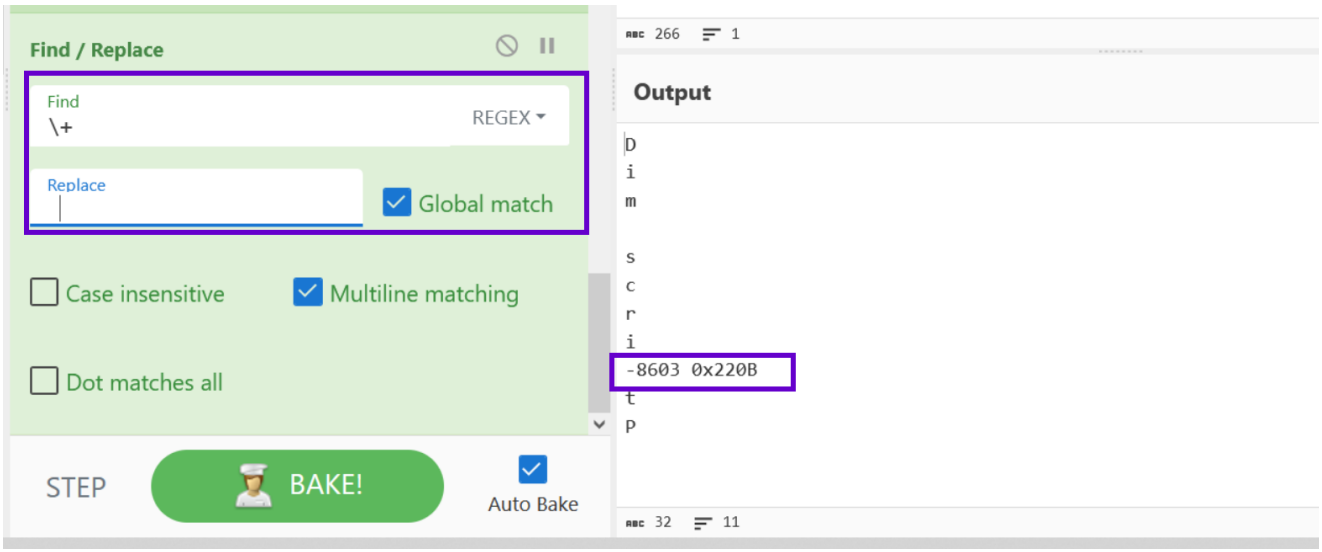
After leaving the Subsection for division, we can create a new Subsection specifically for Addition.

We can do this with another regular expression `-?\w+\+\w+`. This regular expression accounts for the negative values which may be present.



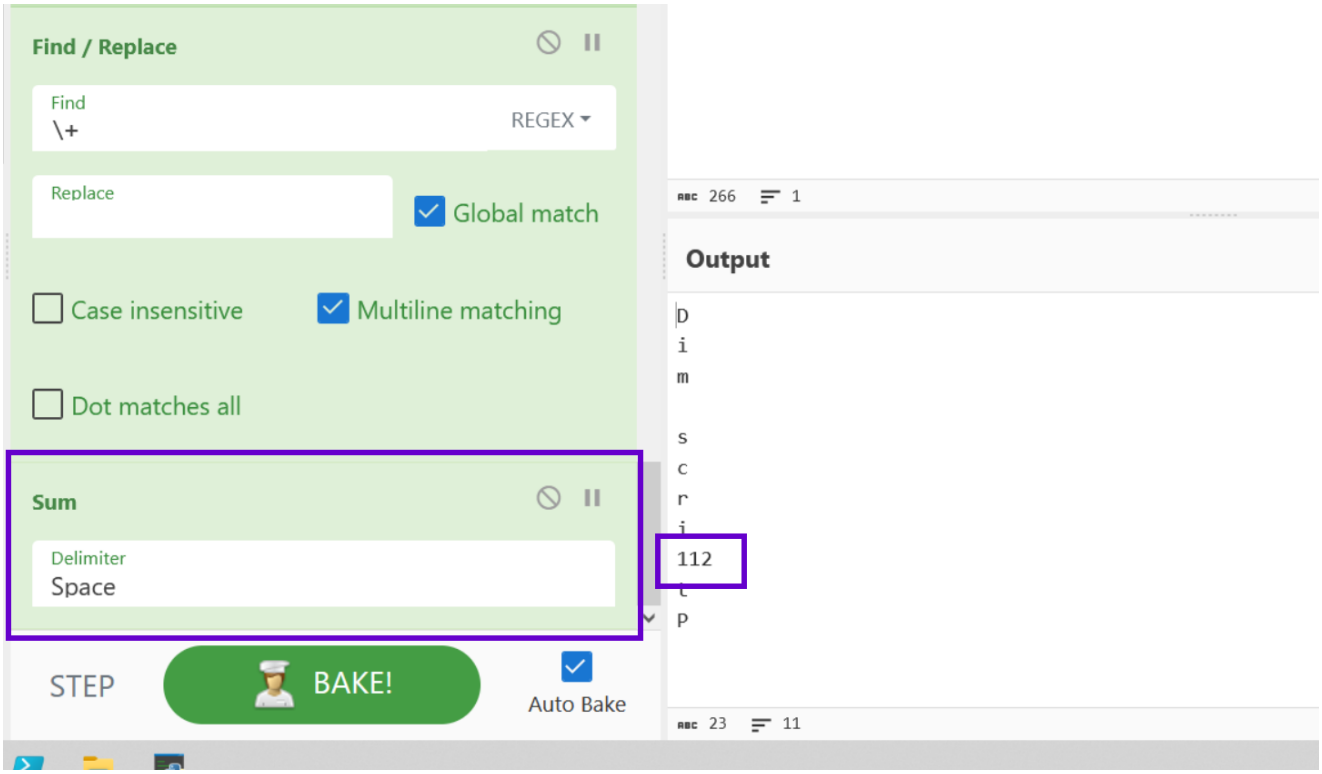
Similar to the division operation, we need to remove the `+` operators and turn the lines into a list separated by a space.

We can this again with a simple `Find/Replace`



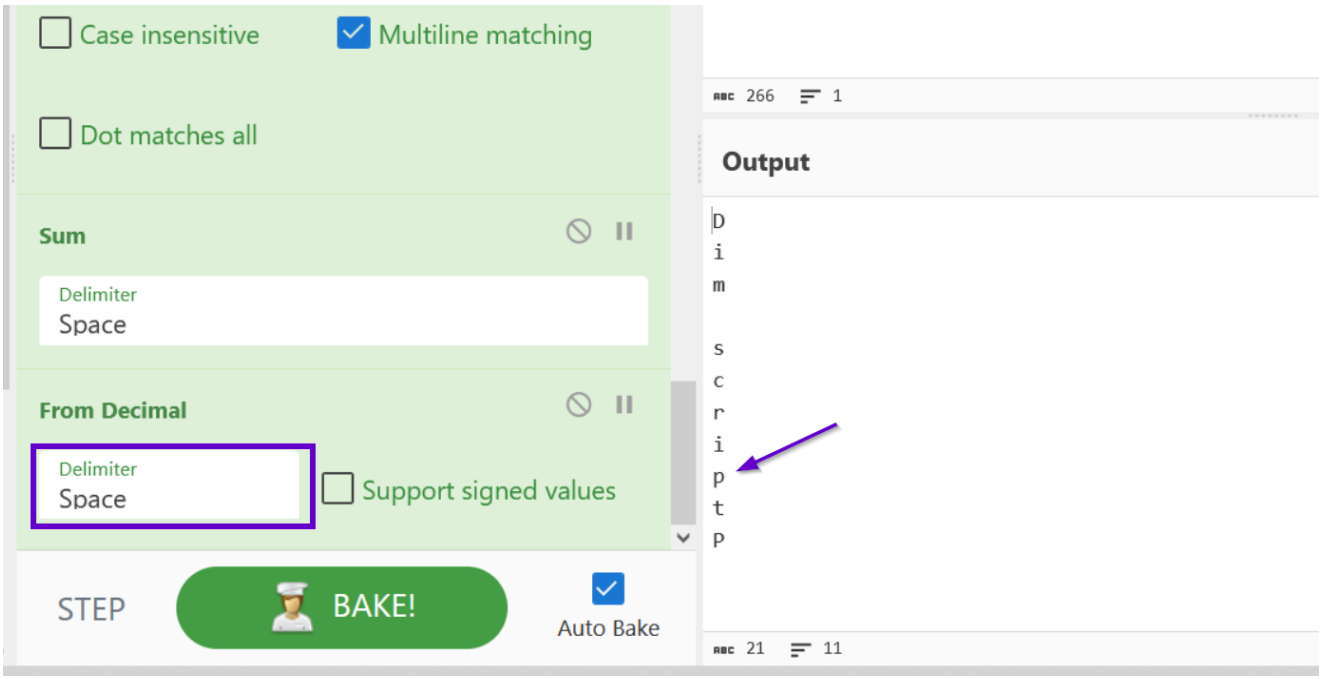
Now that we have a clean list for our addition lines, we can apply a `SUM` operation.

This will add the values together and produce an ASCII charcode.



We can now apply a `From Decimal` operation to obtain the resulting character.

The obfuscated script now looks much better and no longer contains obfuscated content.

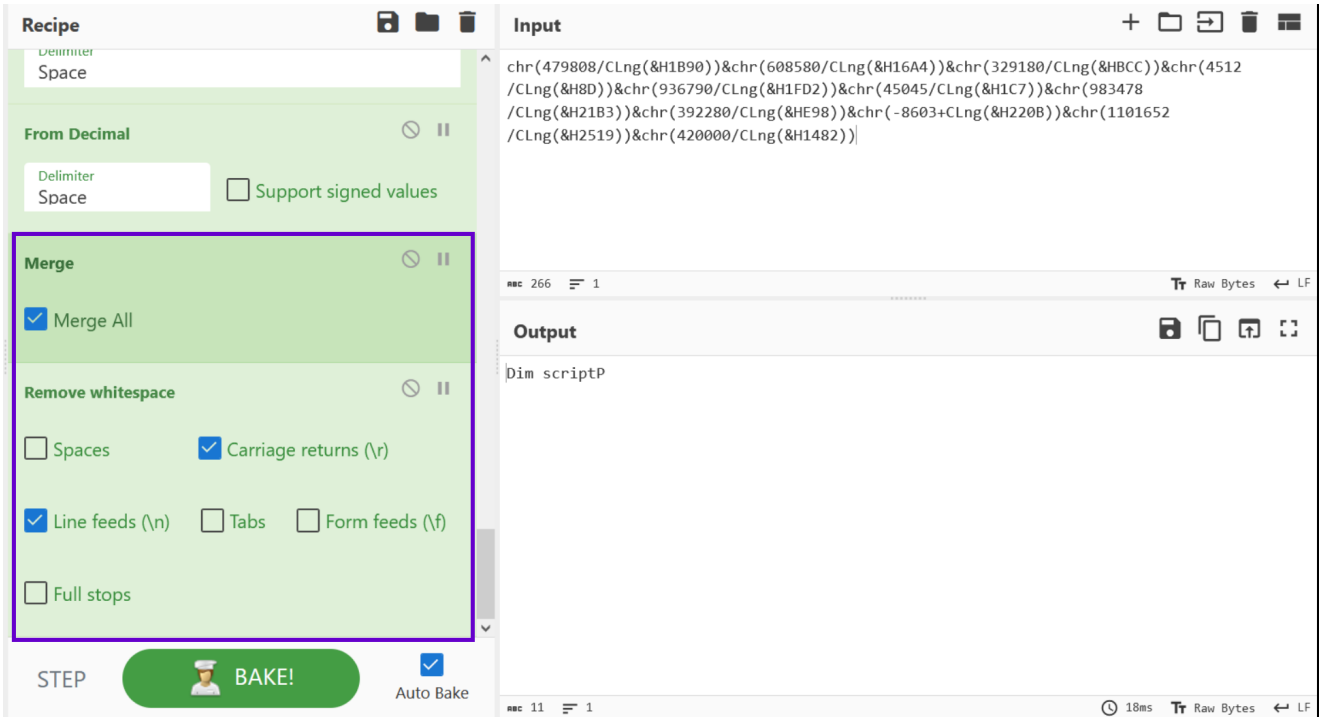


Our deobfuscation prototype is complete, so we can go ahead and remove all subsections and the newlines that separated them.

We can do this with a `Merge -> Merge All` and `Remove Whitespace -> \r + \n`

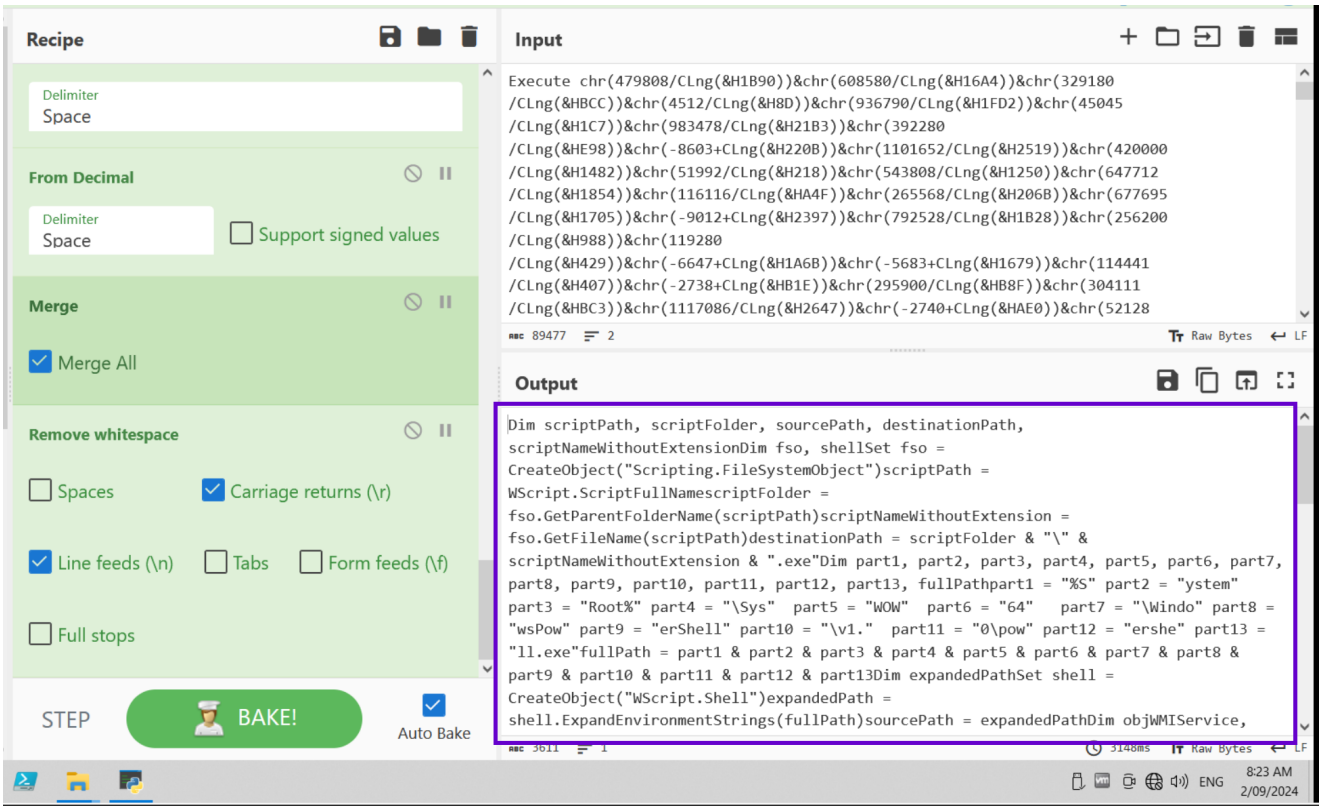
The code output now looks clean and easily readable. So we can go back and add the original obfuscated content.

> Note that we could repeat the previous process if there were other mathematical operations. This script only contains Addition and Division



Reviewing the Final Results

Pasting in the full obfuscated content, we can see the complete deobfuscated result.



We have now deobfuscated line 2 of the initial script. We won't focus on the remainder of the code, but it effectively executes a powershell command that runs a Nanocore payload.

Of interest is that the Nanocore payload is contained in the comments of the initial script.

Since we initially removed these comments, we would need to restore them to obtain the final payload.

Link To The Sample

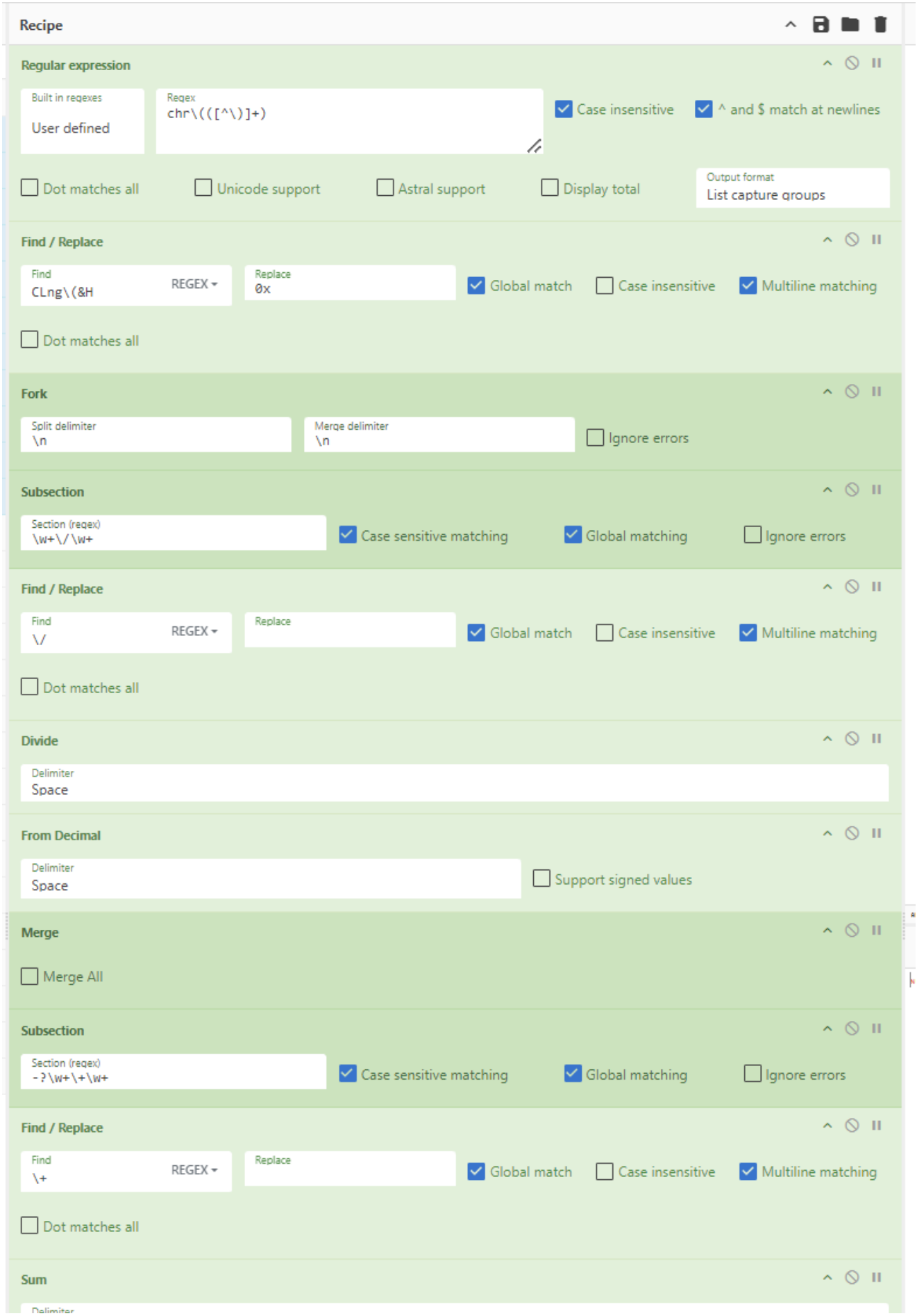
The sample can be found on Malware Bazaar with the following SHA256 and Link.

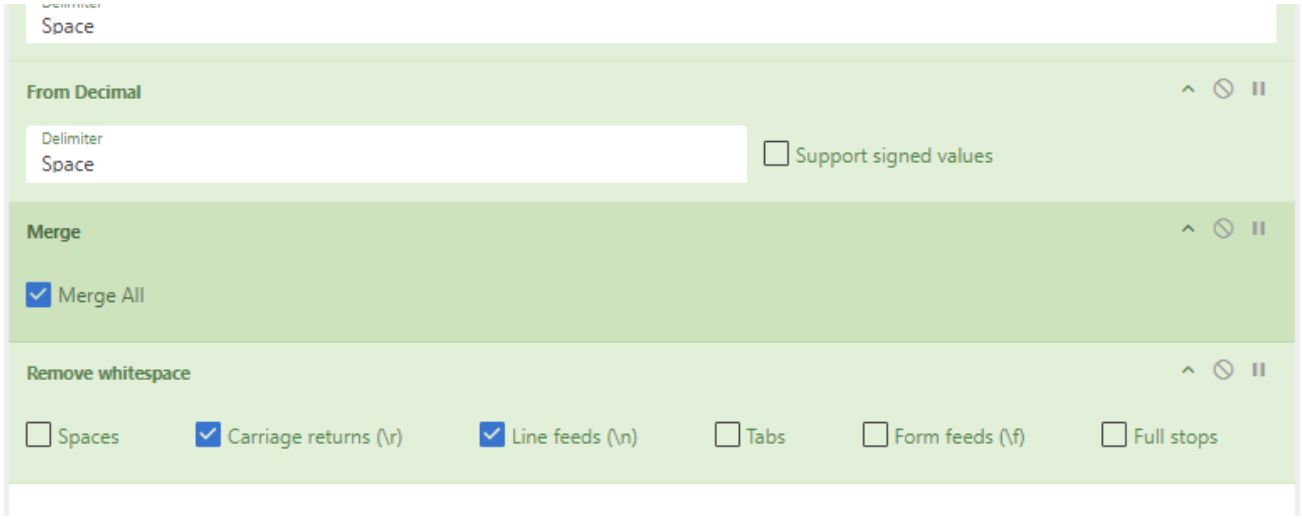
SHA256: `c6092b1788722f82280d3dca79784556df6b8203f4d8f271c327582dd9dcf6e1`

CyberChef Recipe

The complete Cyberchef recipe can be found below.

```
Regular_expression('User defined','chr\\
(([^\\)]+)',true,true,false,false,false,false,'List capture groups')
Find_/_Replace({'option':'Regex','string':'CLng\\(&H'},'0x',true,false,true,false)
Fork('\\n','\\n',false)
Subsection('\\w+\\/\\w+',true,true,false)
Find_/_Replace({'option':'Regex','string':'\\/'},' ',true,false,true,false)
Divide('Space')
From_Decimal('Space',false)
Merge(false)
Subsection('-?\\w+\\+\\w+',true,true,false)
Find_/_Replace({'option':'Regex','string':'\\+'},' ',true,false,true,false)
Sum('Space')
From_Decimal('Space',false)
Merge(true)
Remove_whitespace(false,true,true,false,false,false)
```


Recipe
Regular expression
Built in regexes
User defined
Regex
chr\(([^\)]+)
Case insensitive
^ and $ match at newlines
Dot matches all
Unicode support
Astral support
Display total
Output format
List capture groups
Find / Replace
Find
CLng\(&H
REGEX
Replace
0x
Global match
Case insensitive
Multiline matching
Dot matches all
Fork
Split delimiter
\n
Merge delimiter
\n
Ignore errors
Subsection
Section (regex)
\w+\/\w+
Case sensitive matching
Global matching
Ignore errors
Find / Replace
Find
\/
REGEX
Replace
Global match
Case insensitive
Multiline matching
Dot matches all
Divide
Delimiter
Space
From Decimal
Delimiter
Space
Support signed values
Merge
Merge All
Subsection
Section (regex)
-?\w+\+\w+
Case sensitive matching
Global matching
Ignore errors
Find / Replace
Find
\+
REGEX
Replace
Global match
Case insensitive
Multiline matching
Dot matches all
Sum


Space
From Decimal
Delimiter
Space
Support signed values
Merge
Merge All
Remove whitespace
Spaces
Carriage returns (\r)
Line feeds (\n)
Tabs
Form feeds (\f)
Full stops