Advanced CyberChef Techniques For Malware Analysis - Detailed Walkthrough and Examples

By Matthew

Published: 2024-02-26 · Archived: 2026-04-05 18:46:43 UTC

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.

Examples of Advanced Operations in CyberChef

Before we dive in, let's look at 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 an unknown value that fits a known pattern within the code.

How To Use Registers in CyberChef

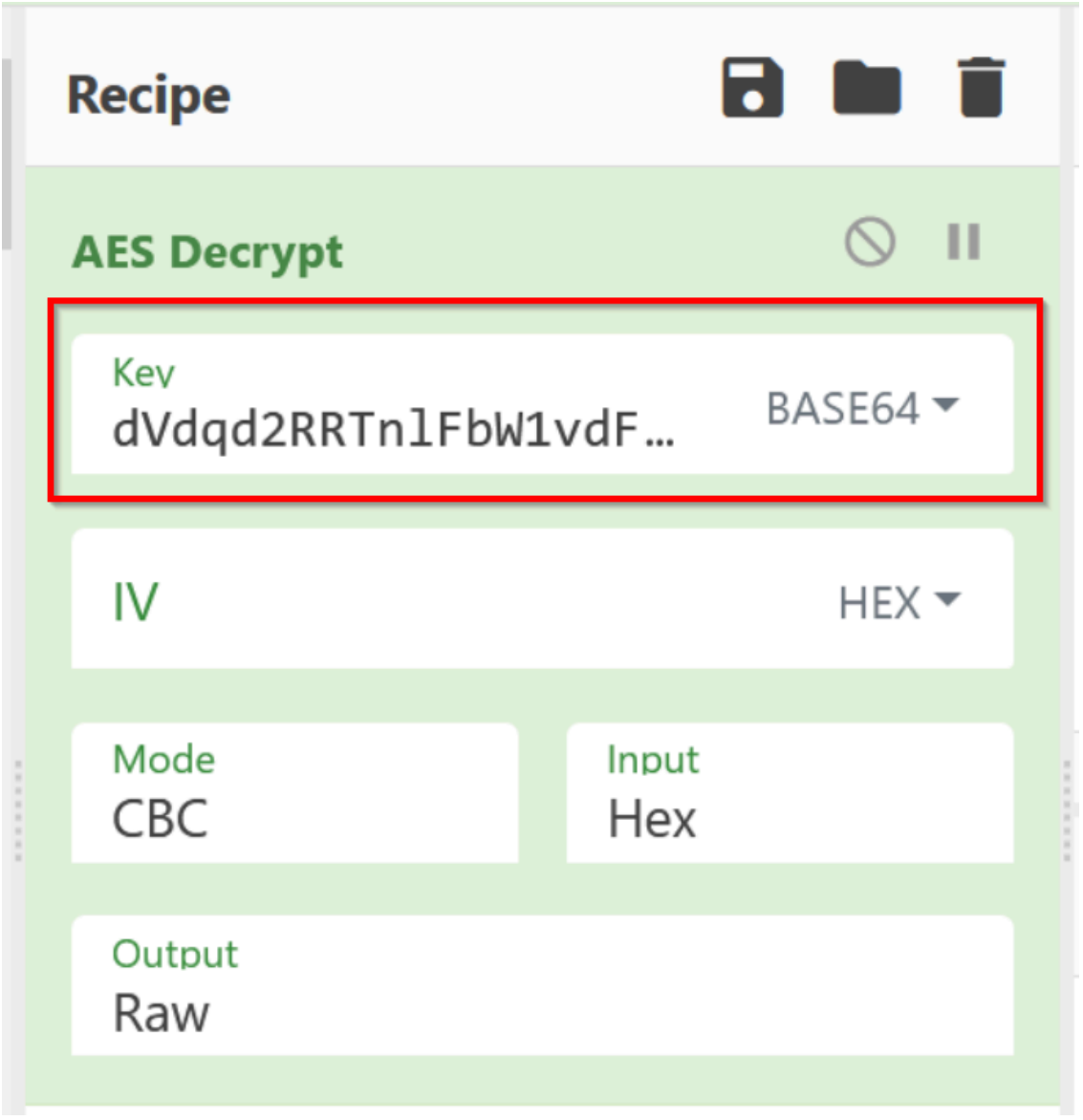
Below we have a Powershell script utilising AES decryption.

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

```
2
3   $Dzxb = 'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVq
4
5   $sUcUPxU = 'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';
6   $GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';
7
8   $GDJXGNY.Mode = [System.Security.Cryptography.CipherMode]::ECB;
9   $GDJXGNY.Padding = [System.Security.Cryptography.PaddingMode]::Zeros;
10  $GDJXGNY.BlockSize = 128;
11  $GDJXGNY.KeySize = 256;
12  $GDJXGNY.Key = [System.Convert]::FromBase64String($sUcUPxU);
13  $KPwOZ = [System.Convert]::FromBase64String($Dzxb);
14  $LdCzhhCP = $KPwOZ[0..15];
15  $GDJXGNY.IV = $LdCzhhCP;
16  $wqkPqQHMD = $GDJXGNY.CreateDecryptor();
17  $EcXHdQkBU = $wqkPqQHMD.TransformFinalBlock($KPwOZ, 16, $KPwOZ.Length - 16);
18  $GDJXGNY.Dispose();
```

AES Decryption Key

We can see the key copied into an AES Decrypt operation.



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

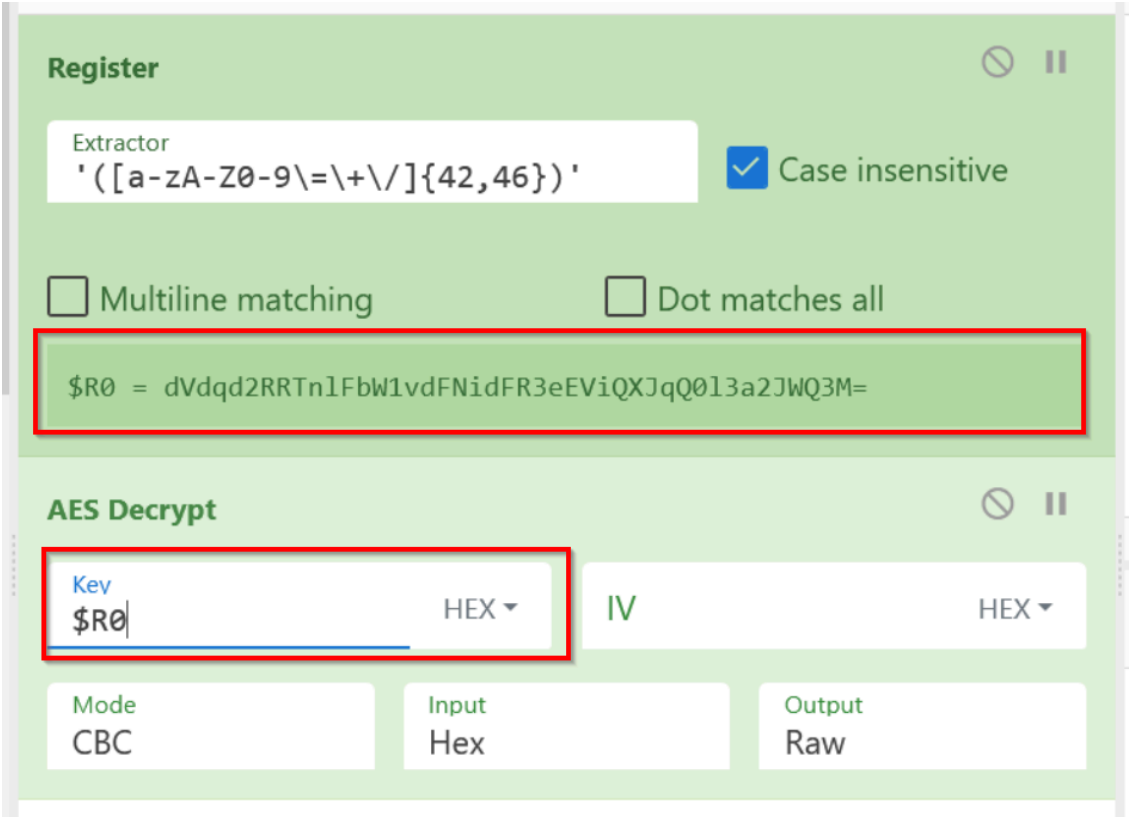
If another sample utilises a different key, then this new key will need to be manually updated for the CyberChef recipe to work.

Registers Example 1

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

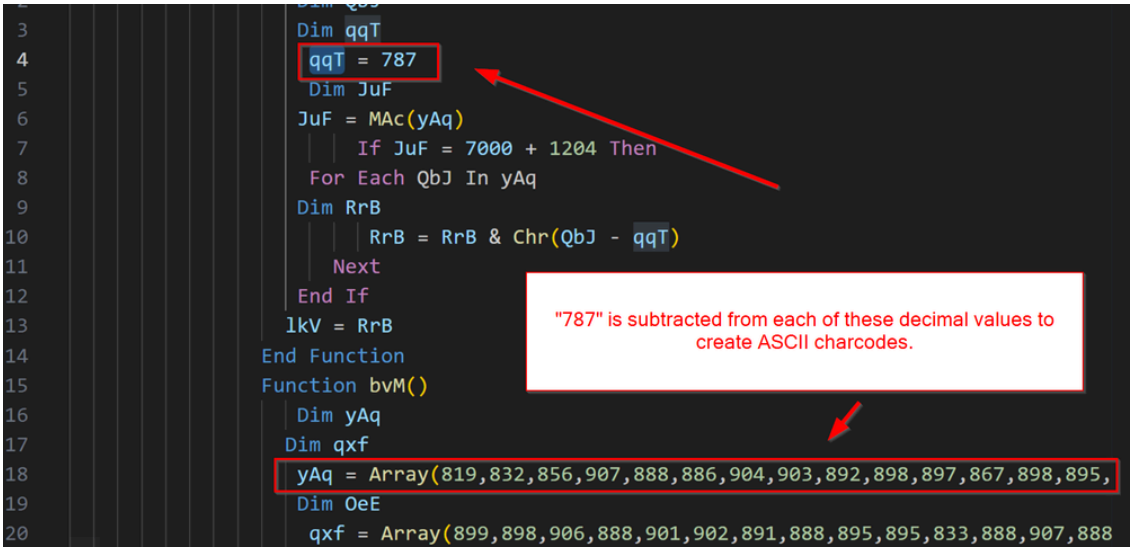
The AES key, in this case, is a 44-character base64 string, hence we can use a base64 regular expression of 44-46 characters to extract the AES Key.

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



Registers Example 2

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

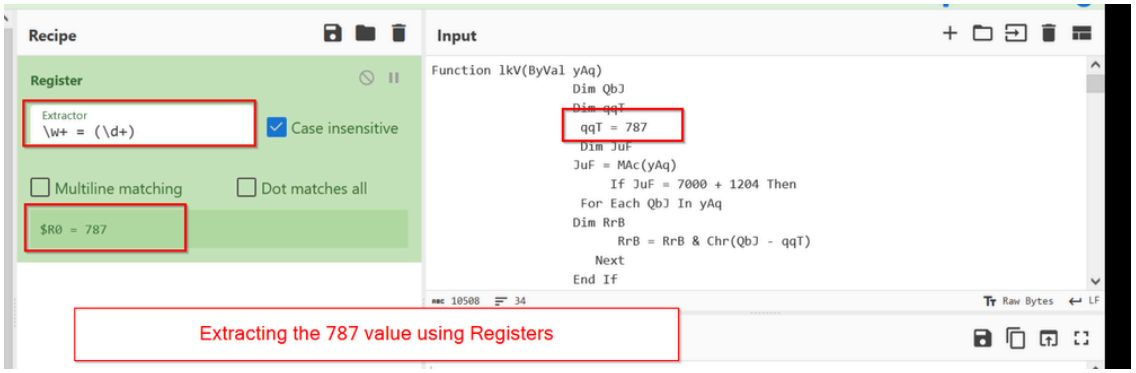


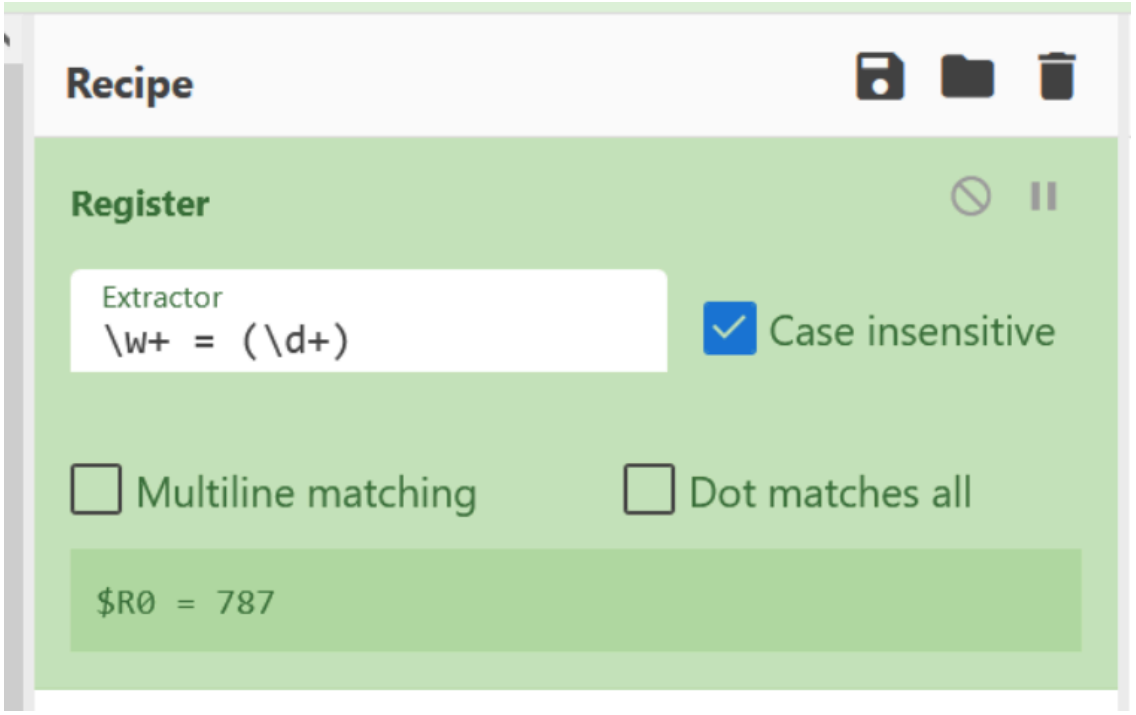
Traditionally, this would be decoded by manually copying out the 787 value and applying this to a subtract operation.

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

A better method is to create another register with a regular expression that matches 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.





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.

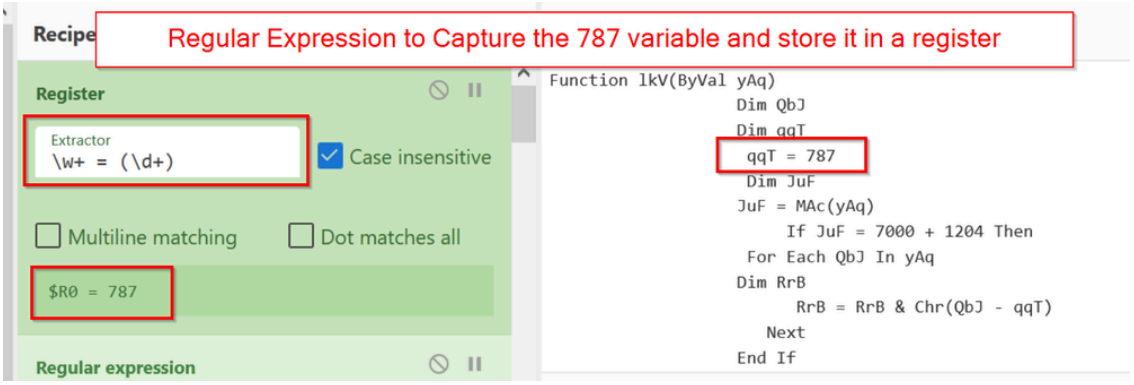
In the development of this configuration extractor, regular expressions are applied in 10 separate operations.

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 a key value used later as part of the deobfuscation process.

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



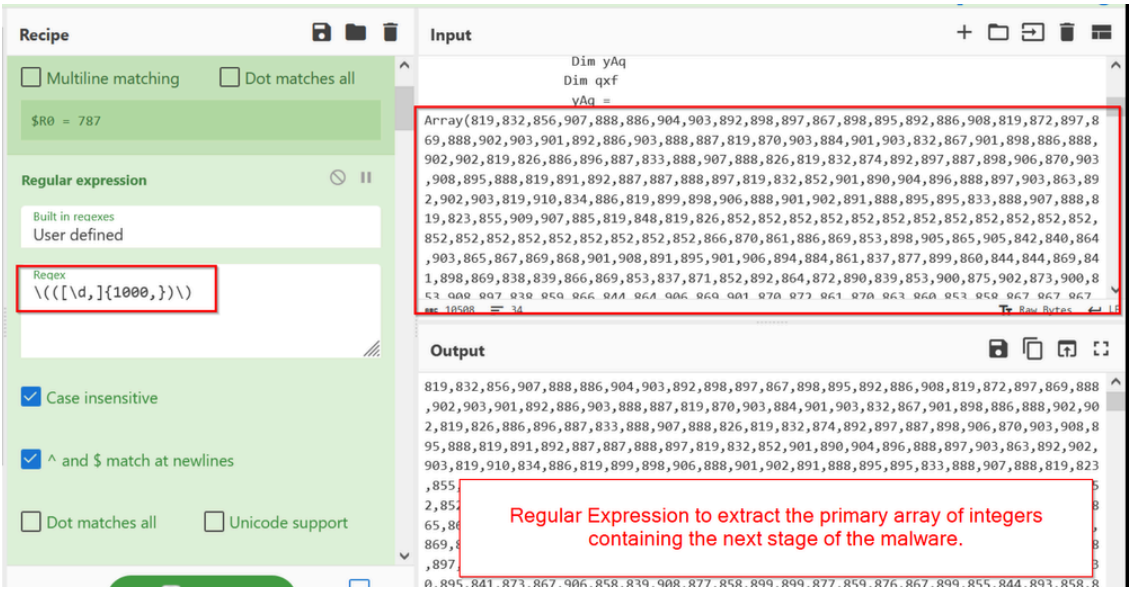
How To Use Regular Expressions to Isolate Text

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 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

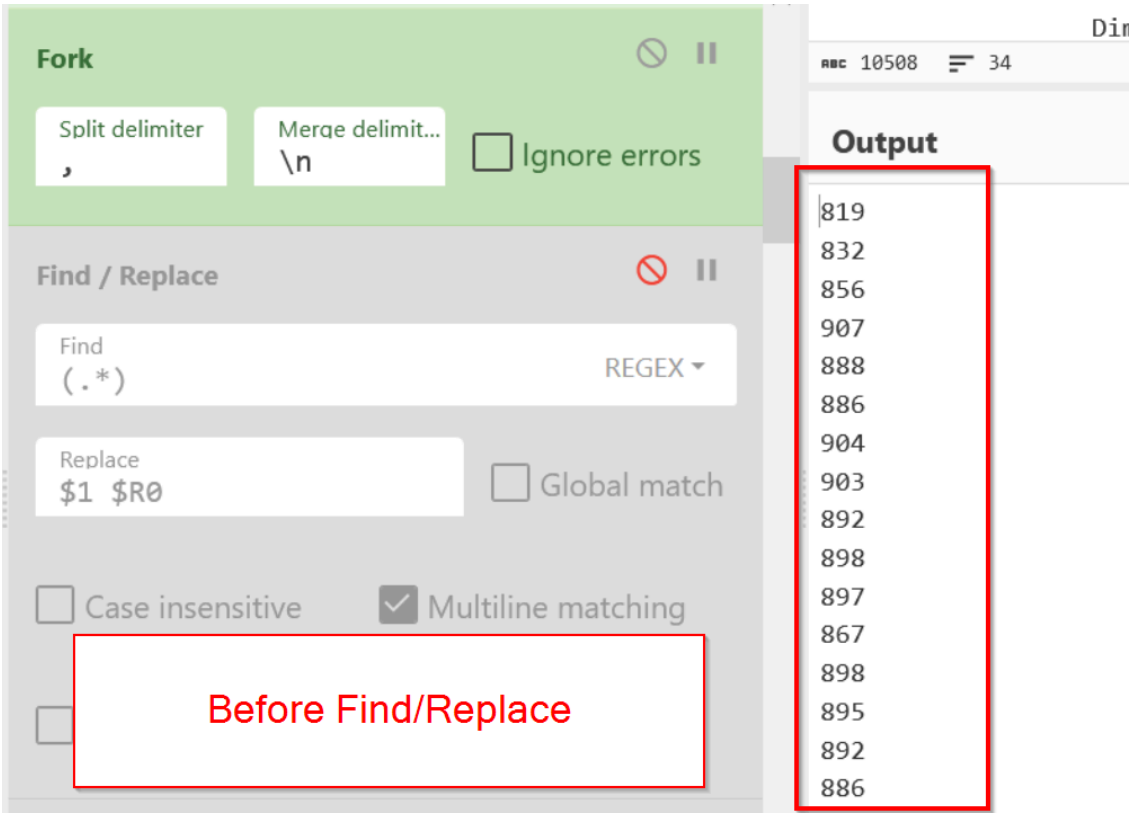


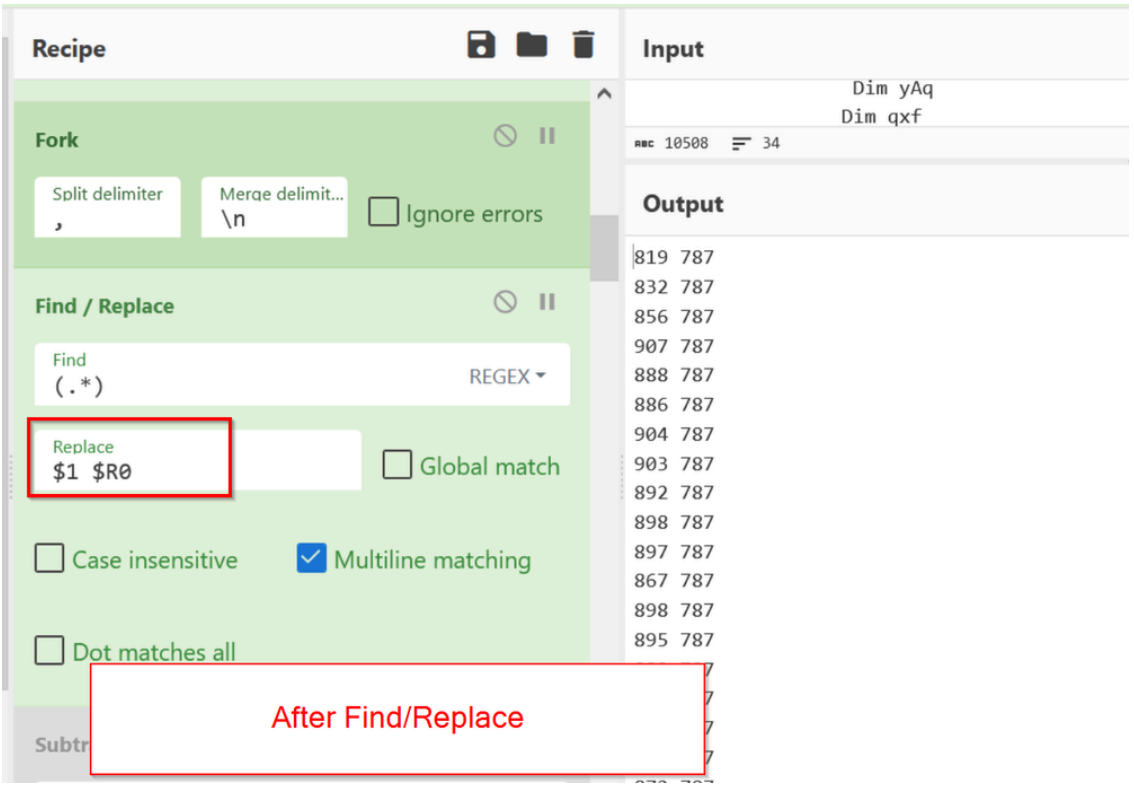
Regular Expressions - Use Case 3 (Appending Values)

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.





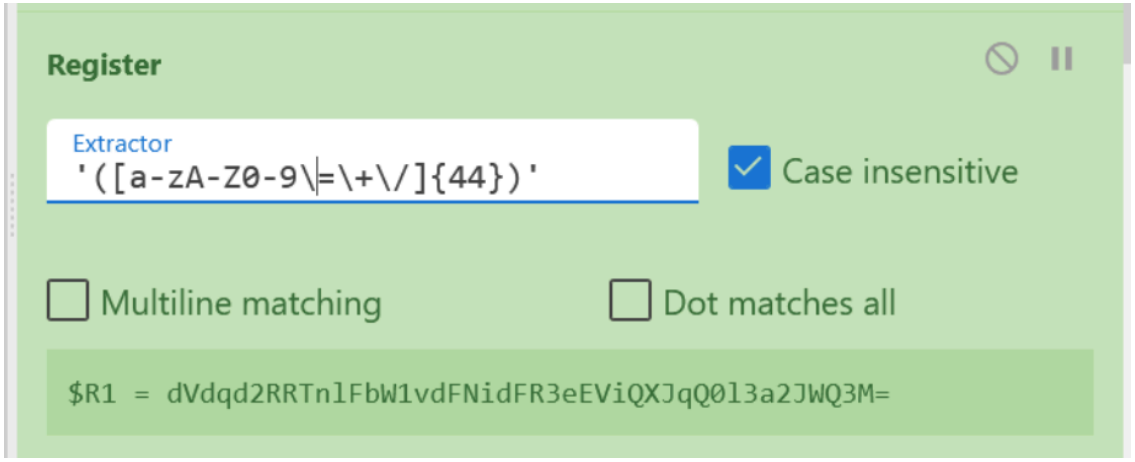
We can utilise regular expressions inside of register operations to extract encryption keys and store these inside of variables.

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 = 'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2T

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

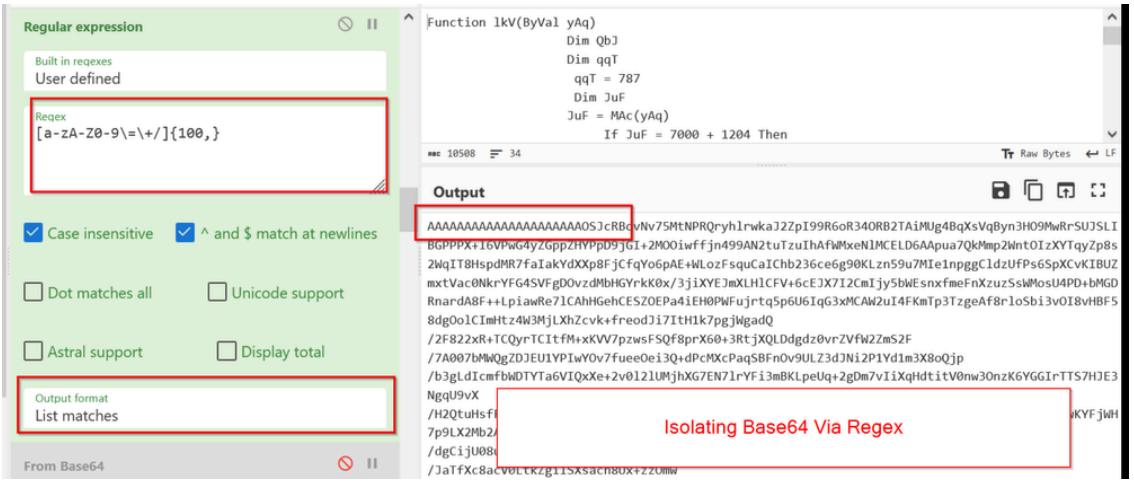


Regular expressions can be used to isolate base64 text containing content of interest.

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 the regex to isolate the primary base64 blob and later perform the AES Decryption.

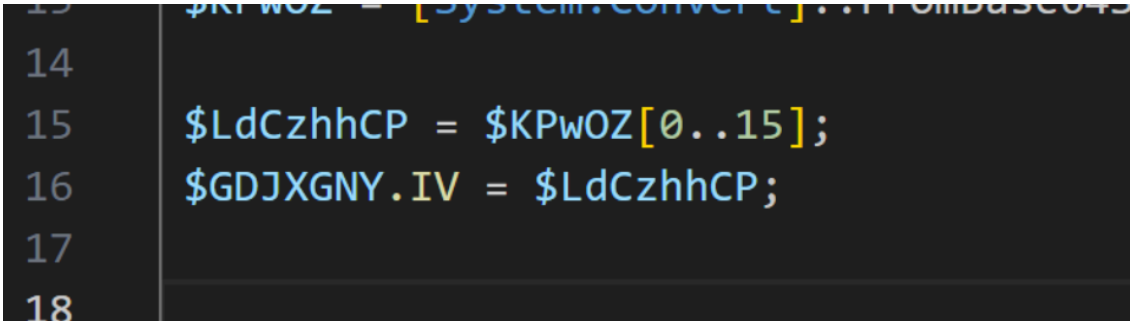
```
2
3   $Dzxb = 'AAAAAAAAAAAAAAAAAAAAAOSJcRBovNv75MtNPRQryhlrwkaJ2ZpI99R6oR34ORB2TAiMUg4BqXsVq
4
5   $sUcUPxU = 'dVdqd2RRTnlFbW1vdFNidFR3eEViQXJqQ0l3a2JWQ3M=';
6   $GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';
7
8   $GDJXGNY.Mode = [System.Security.Cryptography.CipherMode]::ECB;
```

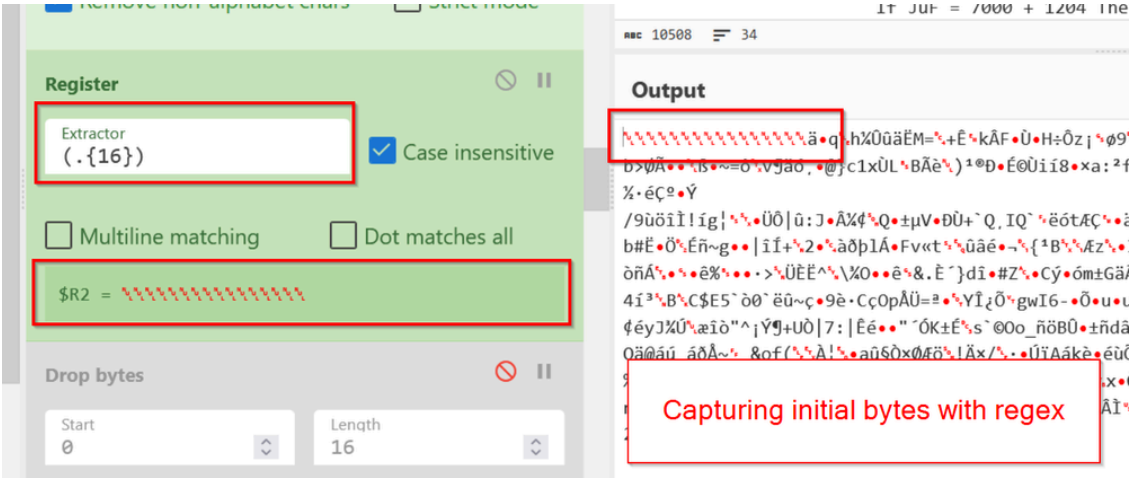


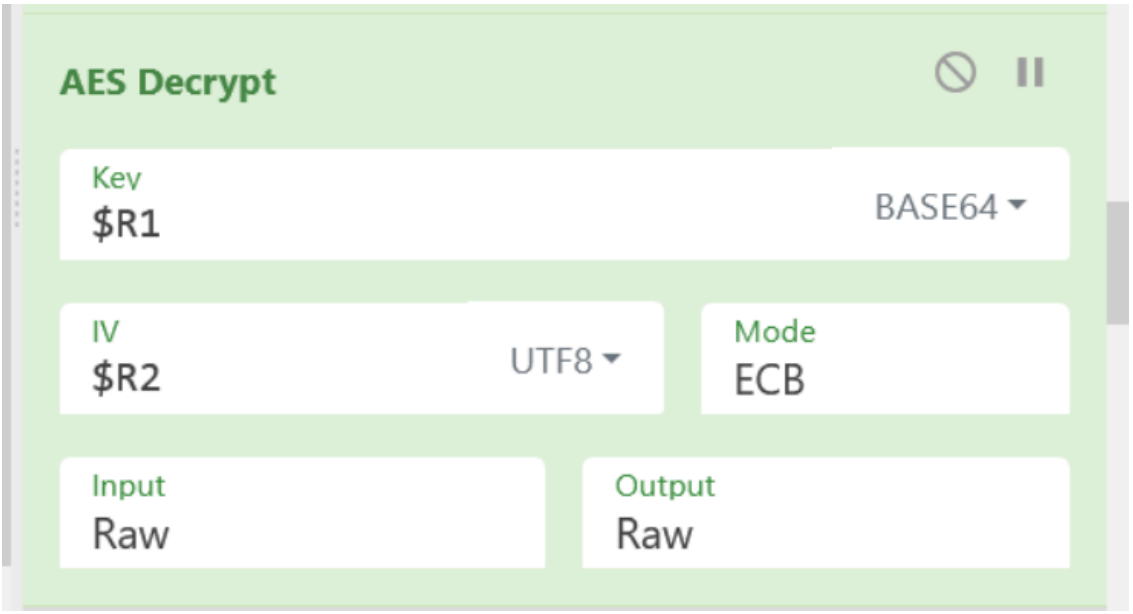
This sample utilises the first 16 bytes of the base64 decoded content to create an IV for the AES decryption.

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.





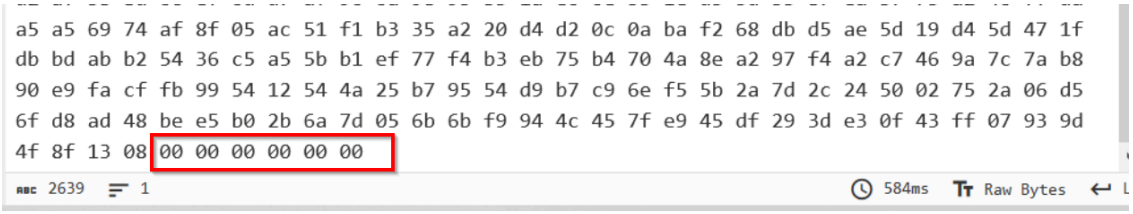


Using Regular Expressions To Remove Trailing Null-Bytes

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

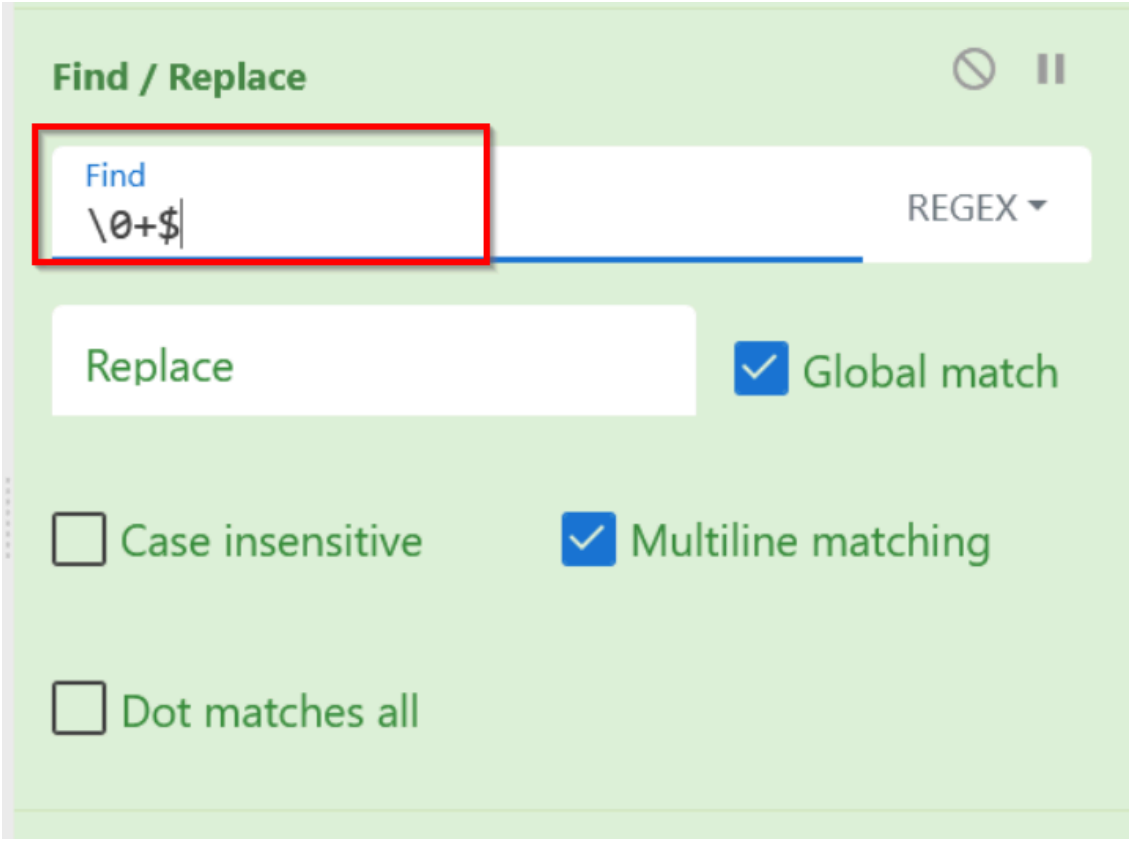
This is particularly useful as sometimes we only want to remove null bytes at the "end" of data. Whereas a traditional "remove null bytes" will remove null bytes everywhere in the code.

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



By applying a null byte search `\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.



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

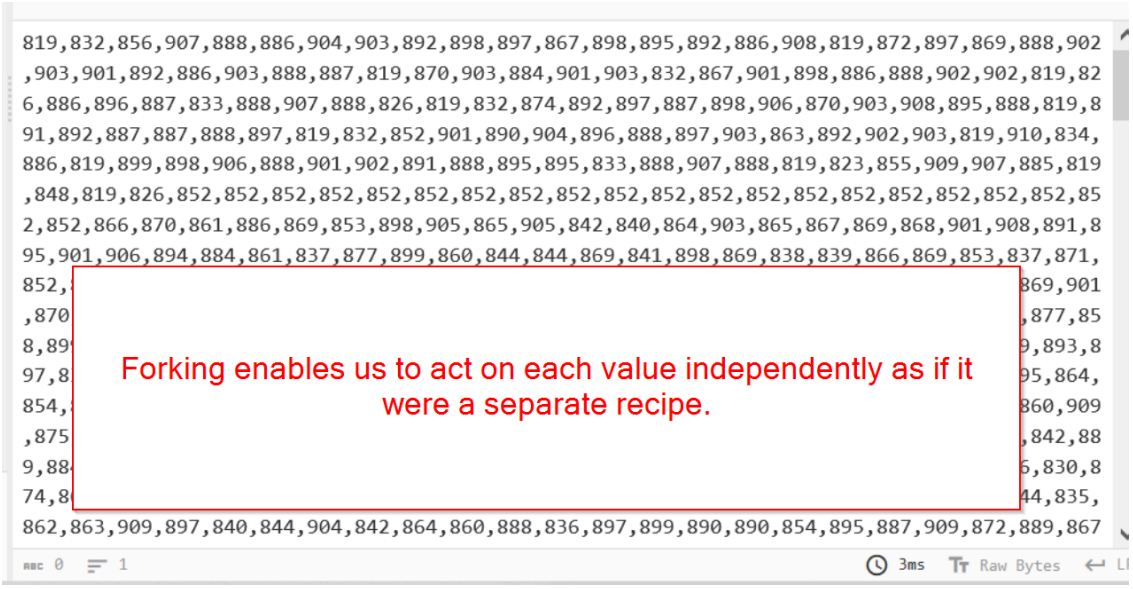
```
db bd ab b2 54 36 c5 a5 5b b1 ef 77 f4 b3 eb 75 b4 70 4a 8e a2 97 f4 a2 c7 46 9a 7c 7a b8
90 e9 fa cf fb 99 54 12 54 4a 25 b7 95 54 d9 b7 c9 6e f5 5b 2a 7d 2c 24 50 02 75 2a 06 d5
6f d8 ad 48 be e5 b0 2b 6a 7d 05 6b 6b f9 94 4c 45 7f e9 45 df 29 3d e3 0f 43 ff 07 93 9d
4f 8f 13 08
```

ABC 2621 1 807ms Raw Bytes

How To Use a Fork in CyberChef

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

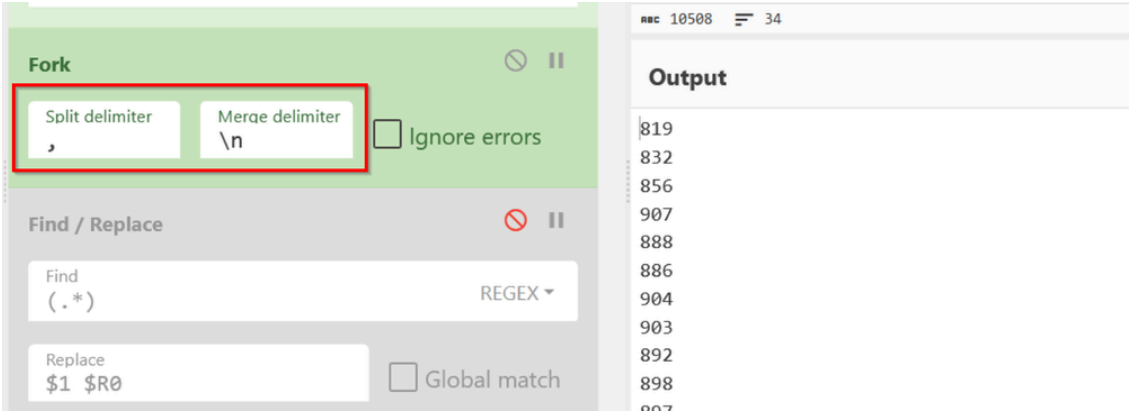
In the use case below, we have a large array of decimal values, and we need to subtract 787 from every single one. A major issue here is that in order to subtract 787, we need to append 787 after every single decimal value in the screenshot below. This would be a nightmare to do by hand.



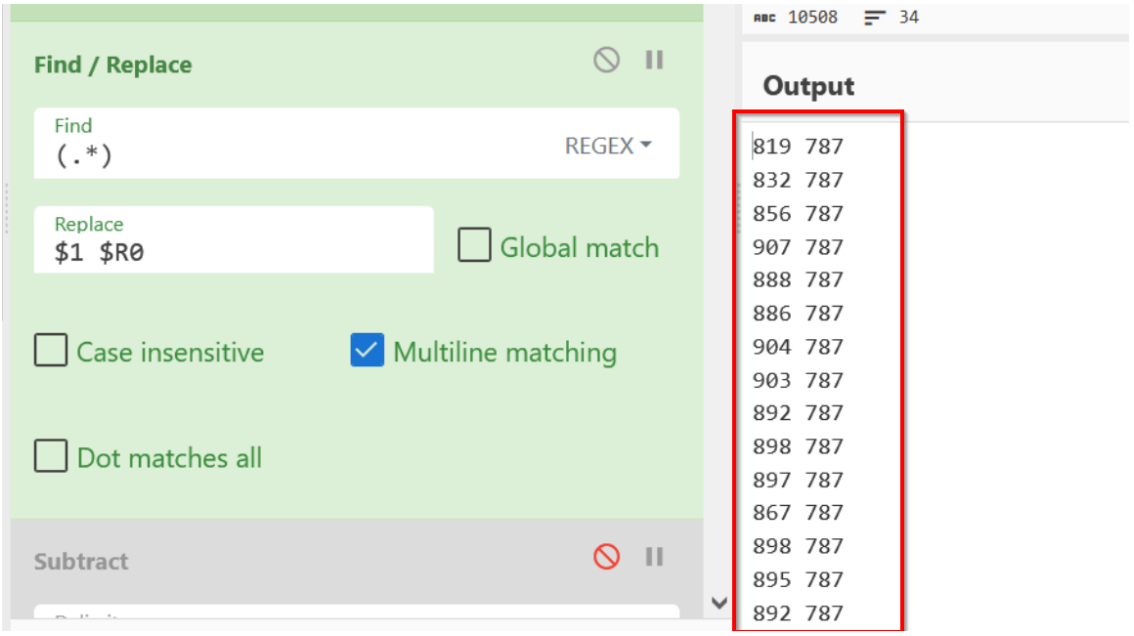
As the data is structured and separated by commas, we can apply a forking operation with a split delimiter of commas and a merge delimiter of newlines.

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

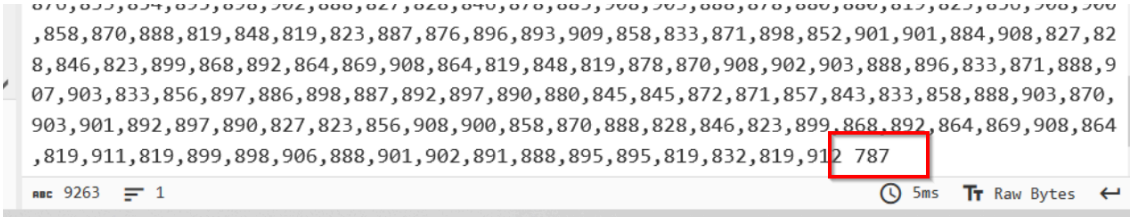
At this point, every new line represents a new input data, and all future operations will act on each line independently.



If we now apply a find-replace operation, we can see that the operation has affected each line individually.



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.



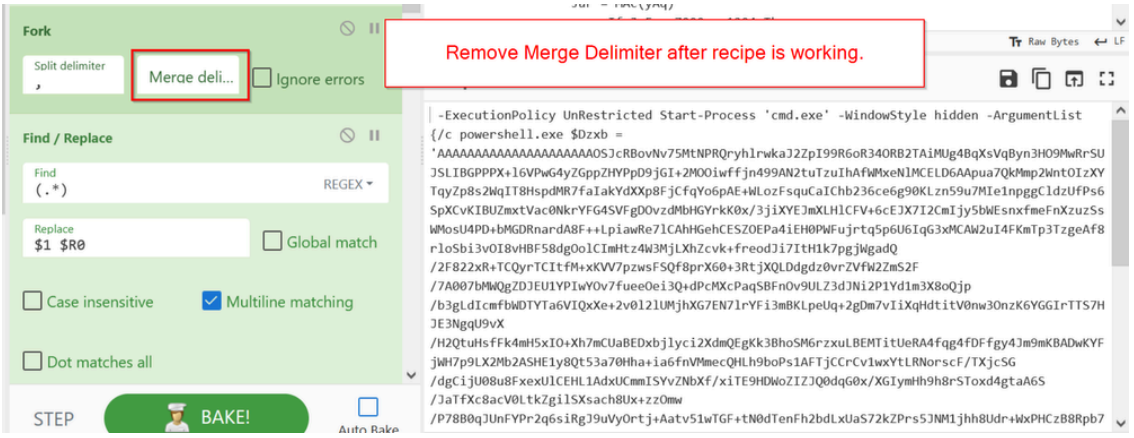
After applying the find/replace, we can continue to apply a subtraction operation and a "From Decimal".

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

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.

We can see the full script after removing the merge delimiter.

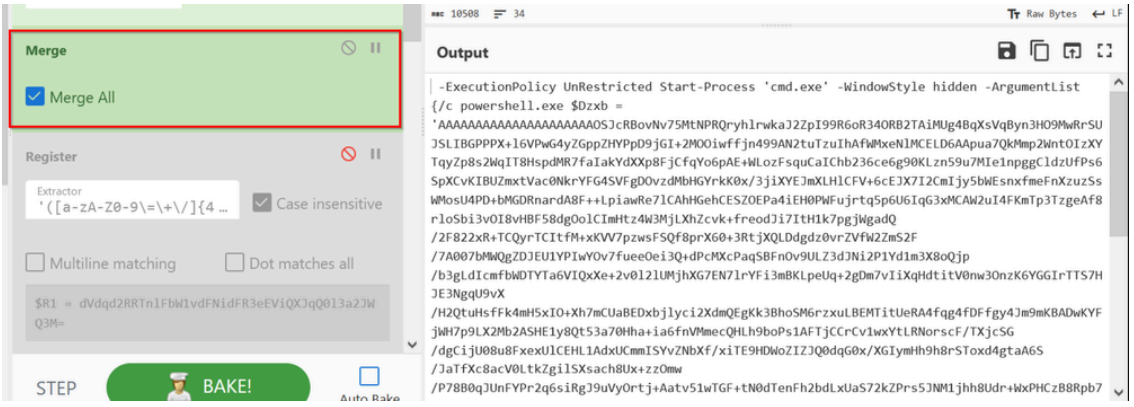


How To Apply a Merge Operation in CyberChef

A `merge` operation is essentially an "undo" for fork operations.

After successfully decoding content using a fork, you should apply a `merge` to ensure that the new content can be analysed appropriately.

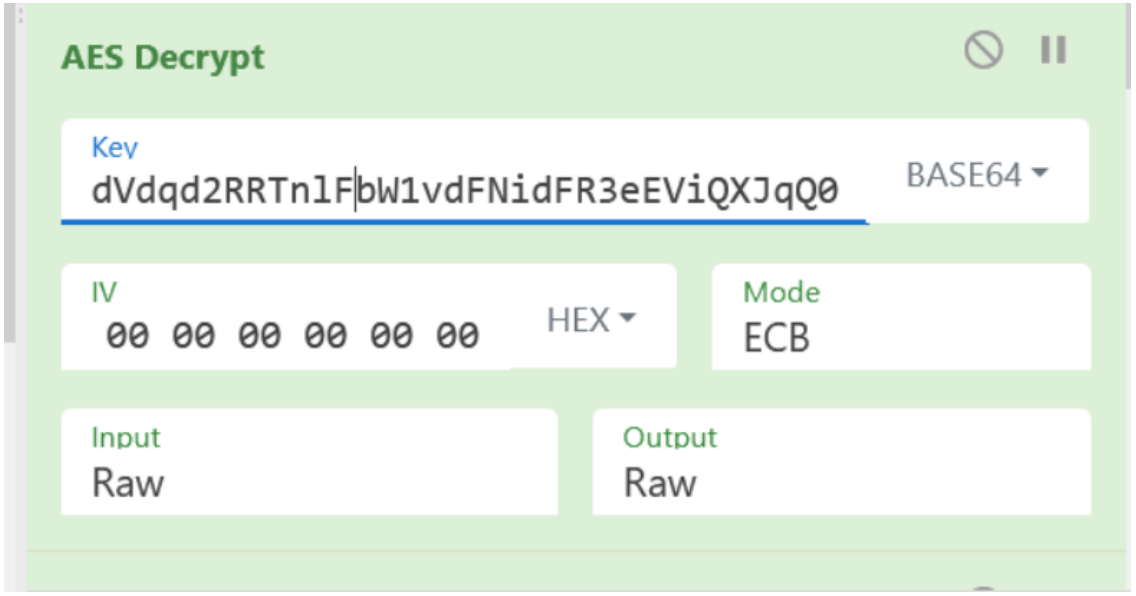
Without a merge, all future operations would affect only a single character and not the entire script.

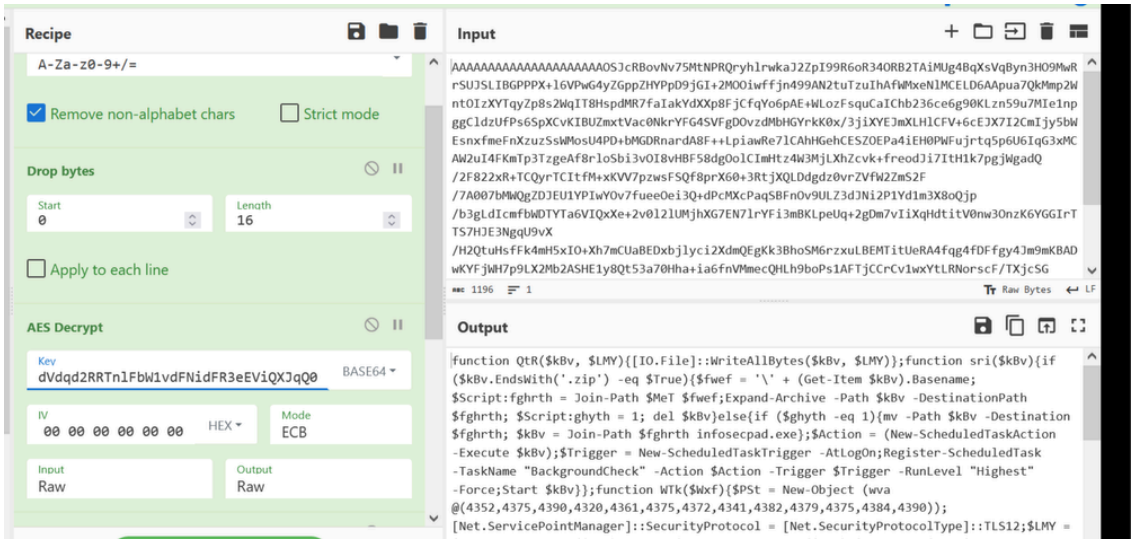


Cyberchef is capable of AES Decryption via the AES Decrypt operation.

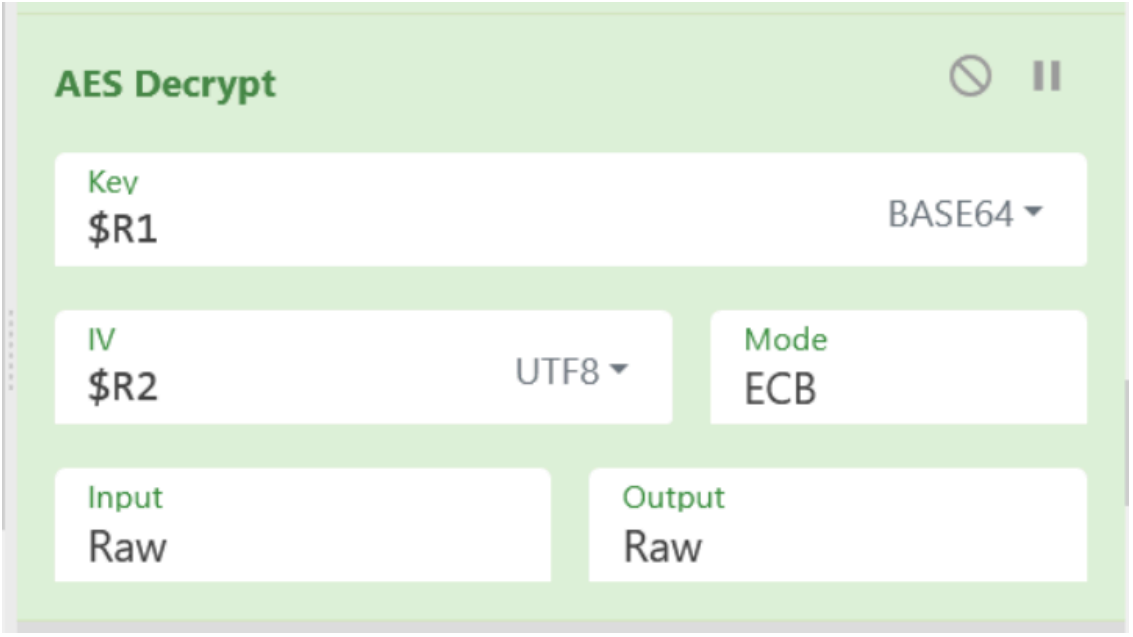
To utilise AES decryption, look for the key indicators of AES inside of malware code and align all the variables with the AES operation.

For example, align the Key, Mode, and IV. Then, plug these values into CyberChef.



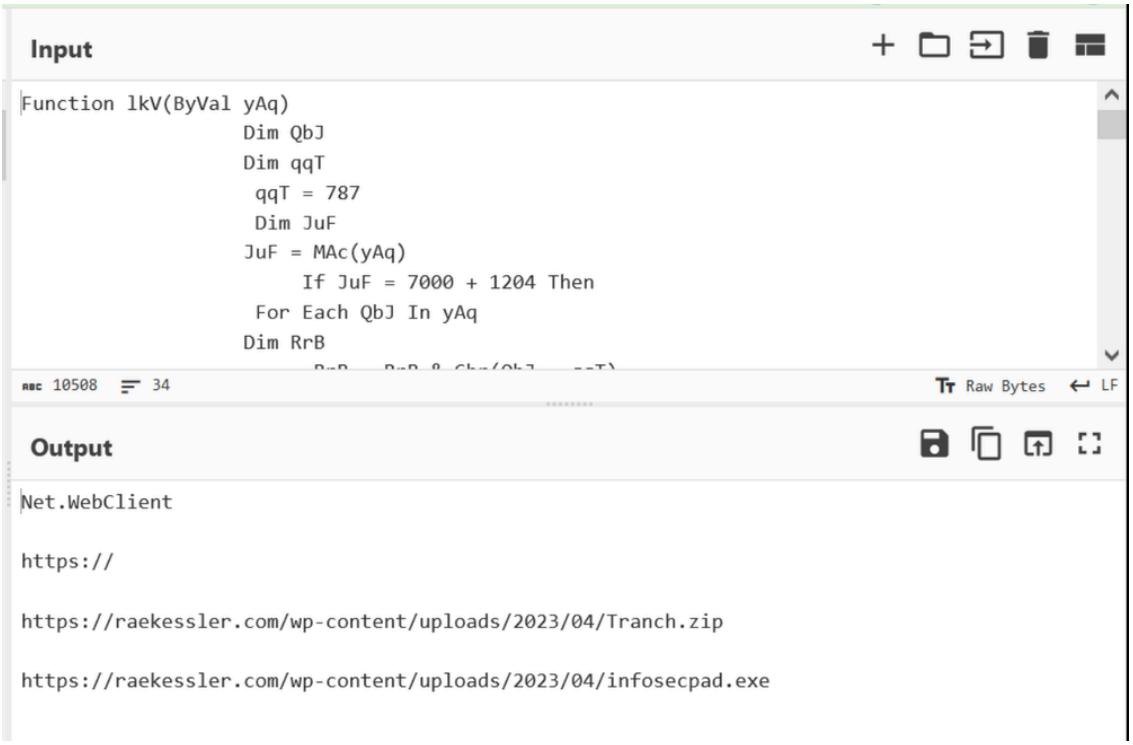


Eventually, you can effectively automate this using Regular Expressions and Registers, as previously shown.



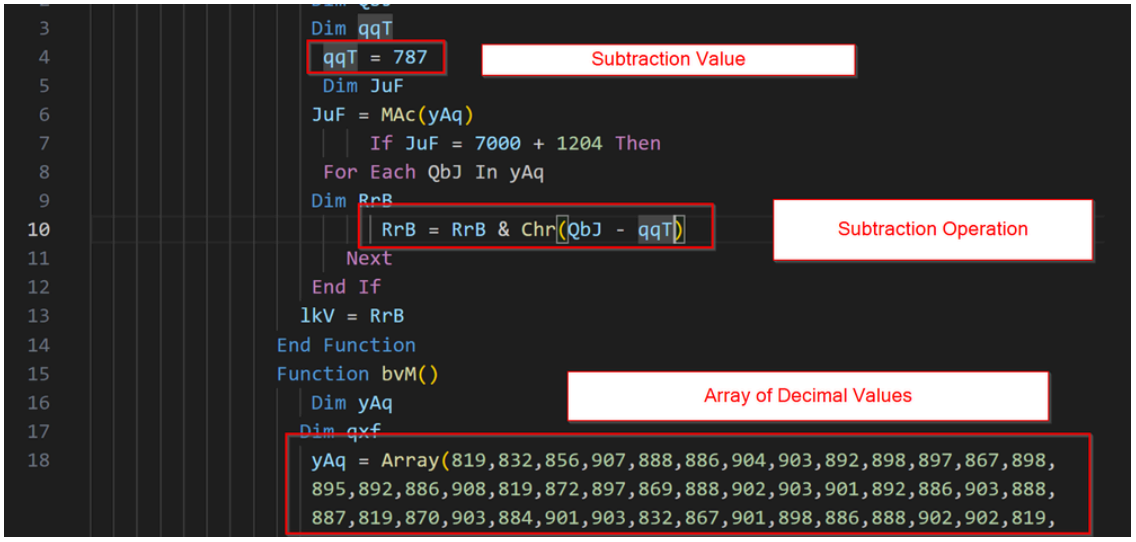
Utilising all of these techniques, we can develop a configuration extractor for a NetSupport Loader with 3 separate scripts that can all be decoded within the same recipe.

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



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.

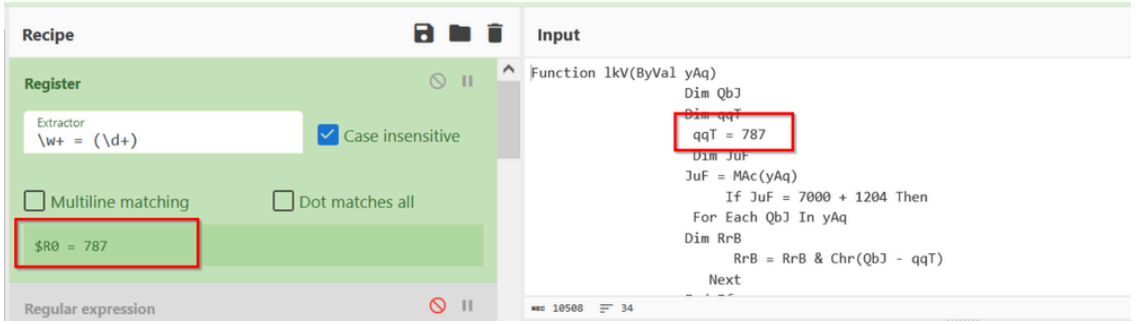


To decode this component, we must

- Use a Register to extract the subtraction value
- Use a Regular Expression to extract the decimal array
- Use Forking to Separate 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 char codes
- Apply a "From Decimal" to produce the 2nd stage
- Use a Merge operation to enable analysis of the 2nd stage script.

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.

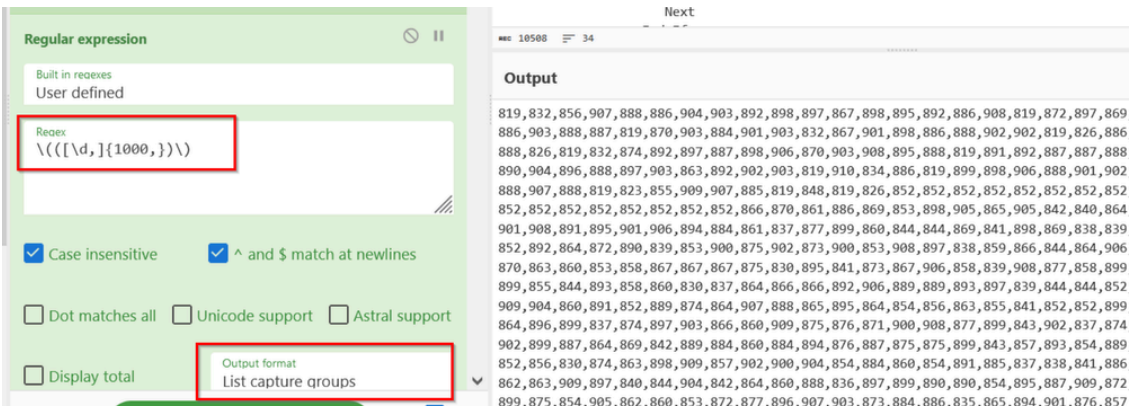


The second step is to extract out the main decimal array using a regular expression and a capture group.

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 `\(` and `\)`.

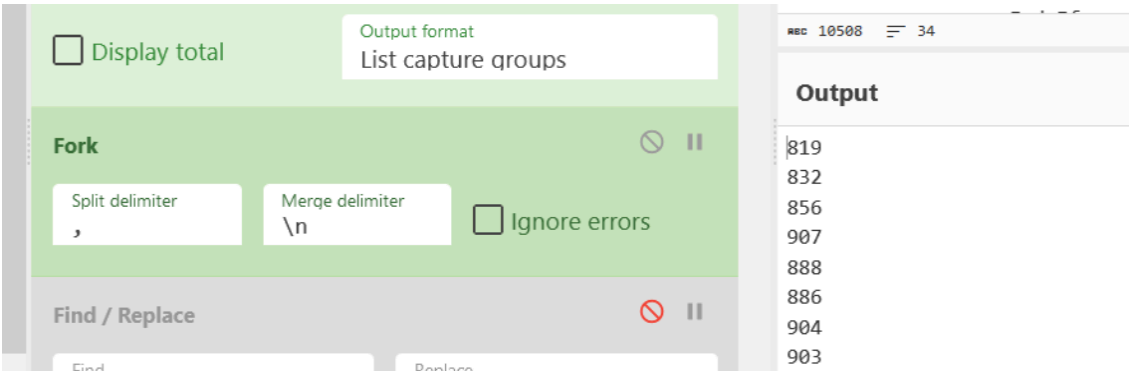
The inner brackets without escapes form the capture group.



Operation 3 - Separating the Decimal Values

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

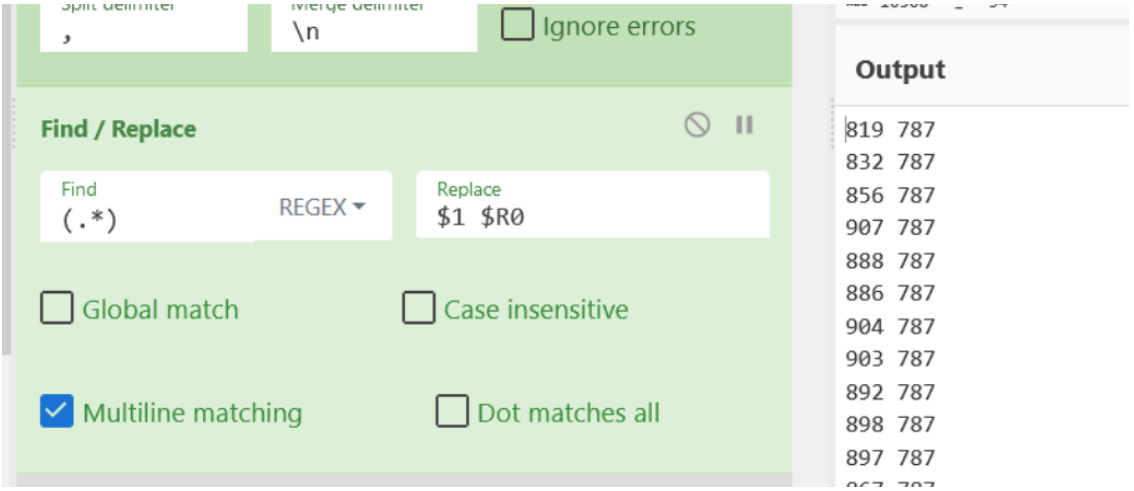
The Fork defines a delimiter at the commas present in the original code, and specifies a Merge Delimiter of `\n` to improve readability.



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.



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 char codes that form the second stage script.

Note that we have specified a space delimiter, as this is what separates our decimal values from our subtraction values in operation 4.

Operation 6 - Decoding ASCII Code In CyberChef

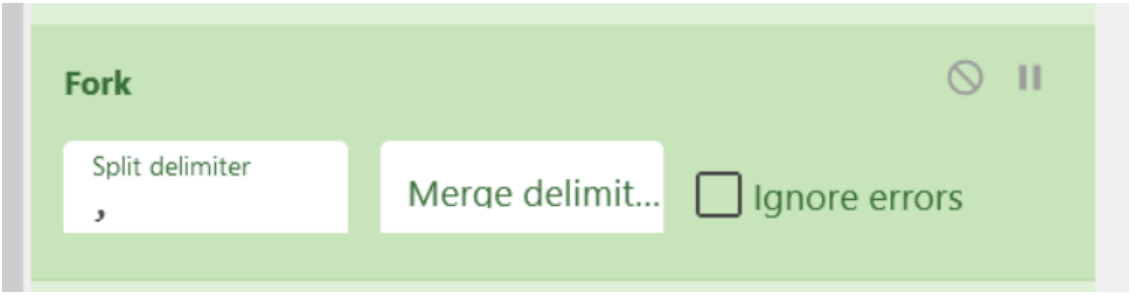
We can now decode the ASCII codes using a "From Decimal" operation.

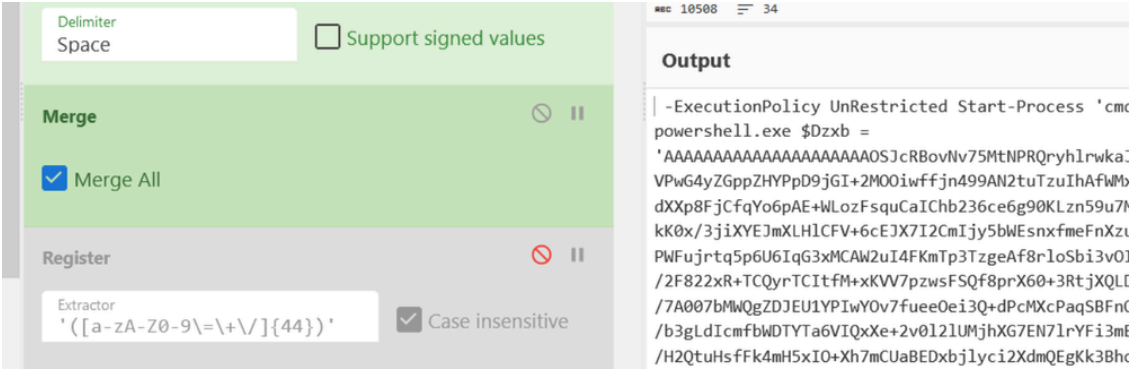
This produces the original script. However, the values are separated via a newline due to our previous Fork operation.

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.

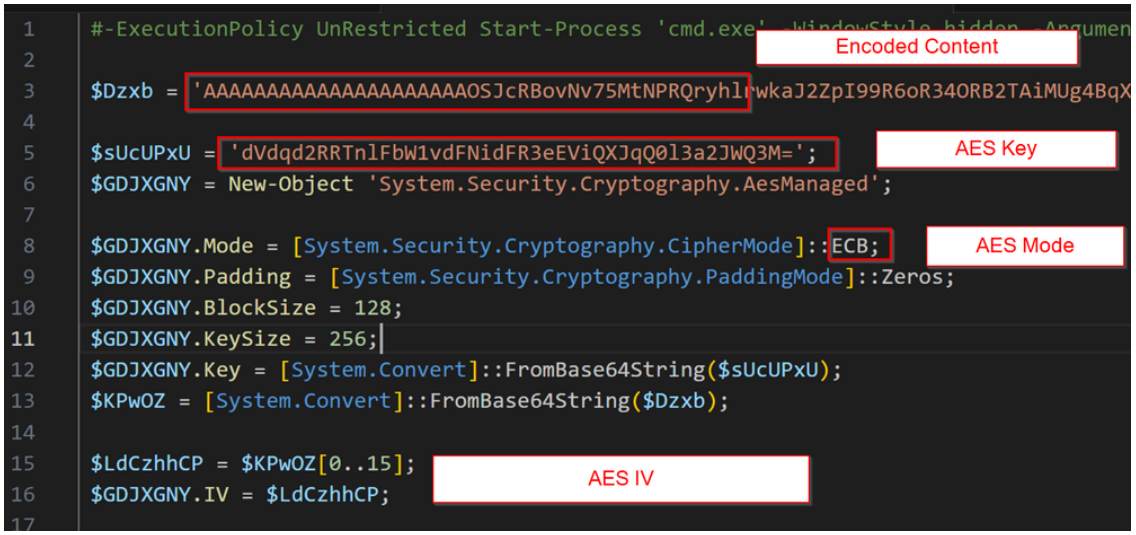




Stage 2 - Powershell Script With AES Encryption (8 Operations)

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.



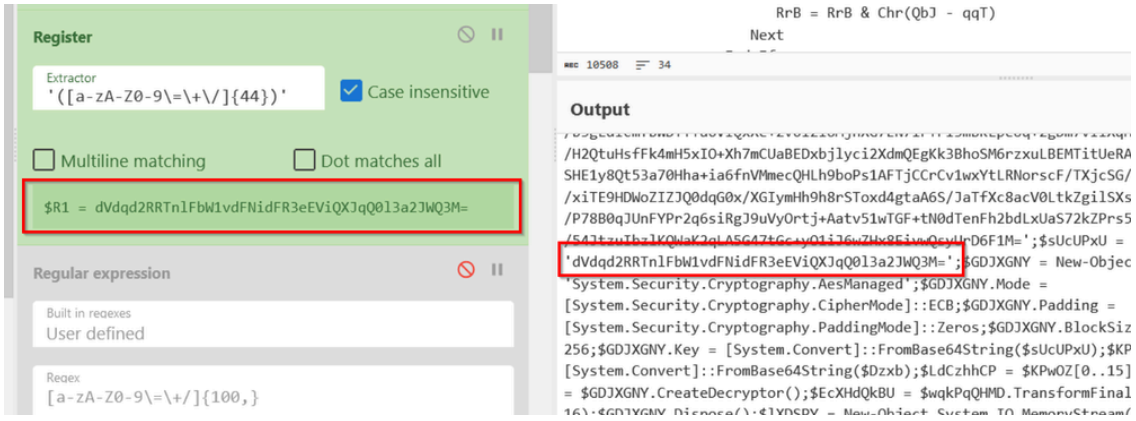
To Decode this stage, we must be able to

- 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

We must now extract the AES Key and store it 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)

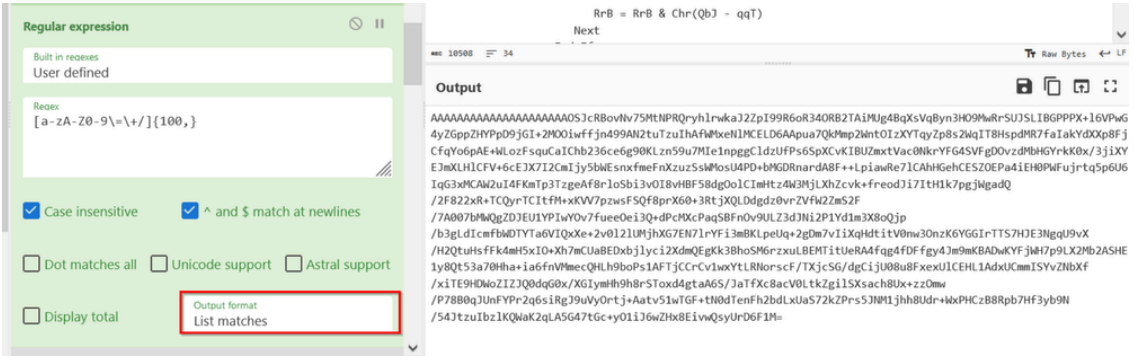
We now have the AES key stored inside of the `$R1` specifying register.



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.

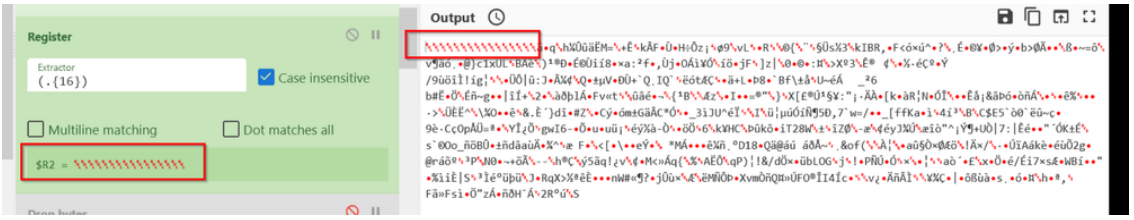


Operation 10 - Decoding The Base64

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

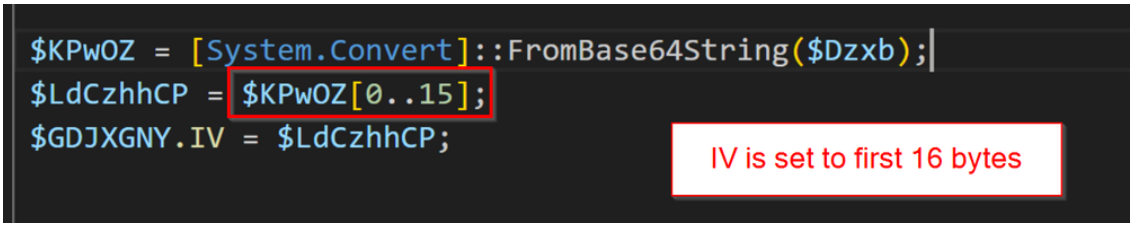
The first 16 bytes of the current data form the initialization vector for the AES decryption.

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



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

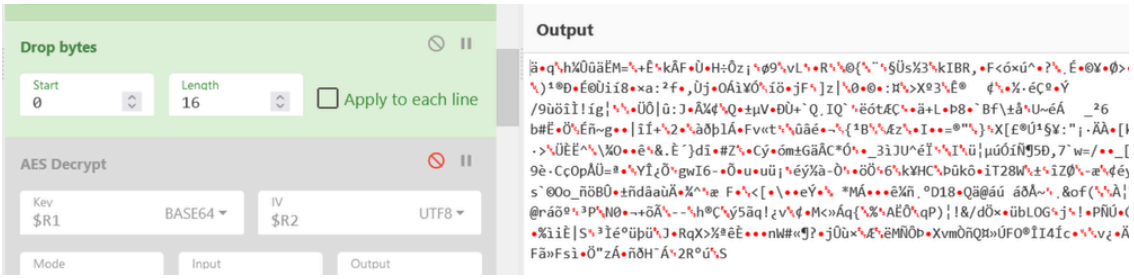
Note how the first 16 bytes are taken after base64 decoding, and then this is set to the IV.



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`,.



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

```
18
19  $wqkPqQHMD = $GDJXGNY.CreateDecryptor();
20  $EcXHdQkBU = $wqkPqQHMD.TransformFinalBlock($KPwOZ, 16, $KPwOZ.Length - 16);
21
22
```

This can be confirmed with the official documentation for TransformFinalBlock.

C# Copy

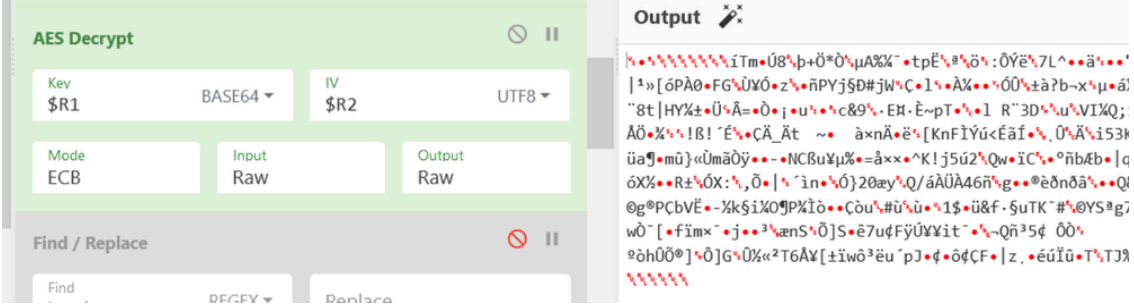
```
public byte[] TransformFinalBlock (byte[] inputBuffer, int inputOffset,
int inputCount);
```

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 when we extracted them



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

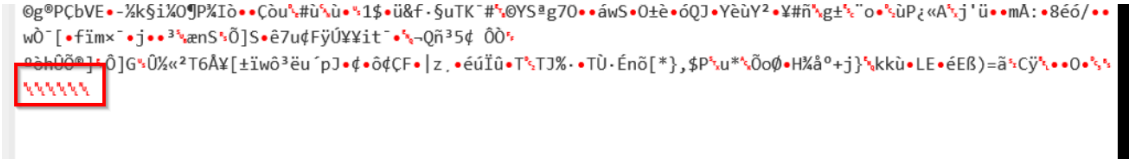
```
$GDJXGNY = New-Object 'System.Security.Cryptography.AesManaged';

$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

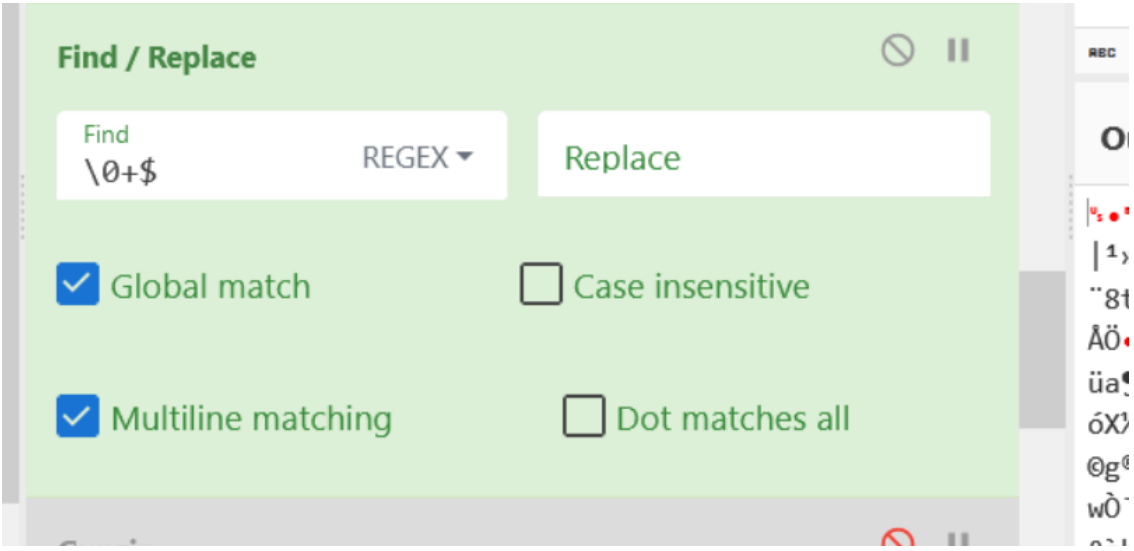
The current data after AES Decryption is compressed using GZIP.

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



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 $"

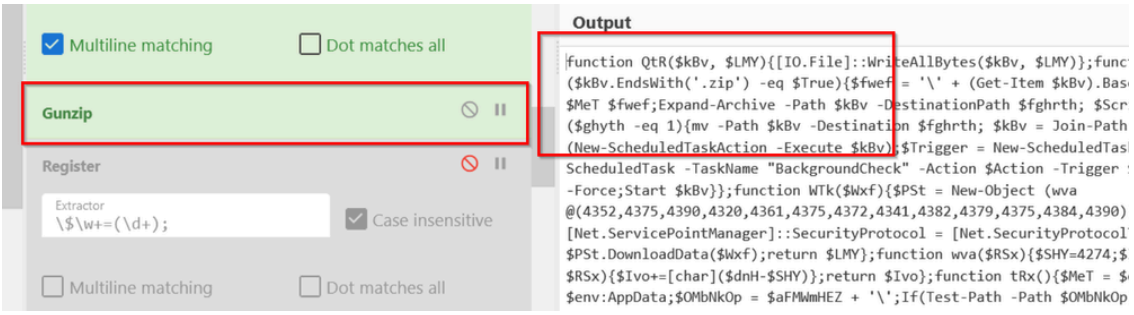
We will leave the "Replace" value empty as we want to remove the trailing null bytes.



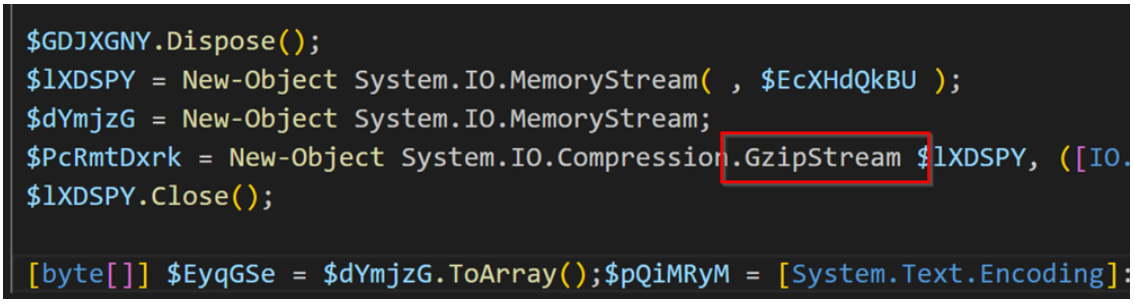
Operation 15 - GZIP Decompression

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.



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

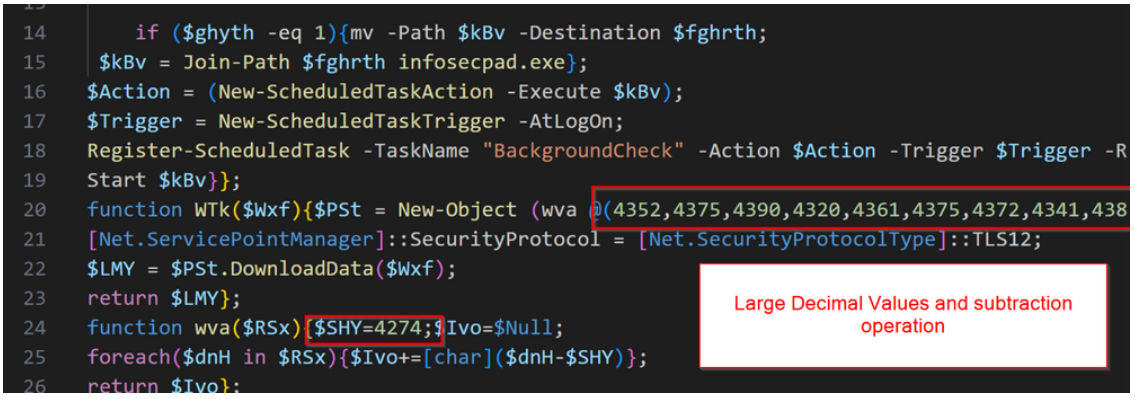


Stage 3 - Powershell Script (7 Operations)

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, with the number 4274 subtracted from each value.

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



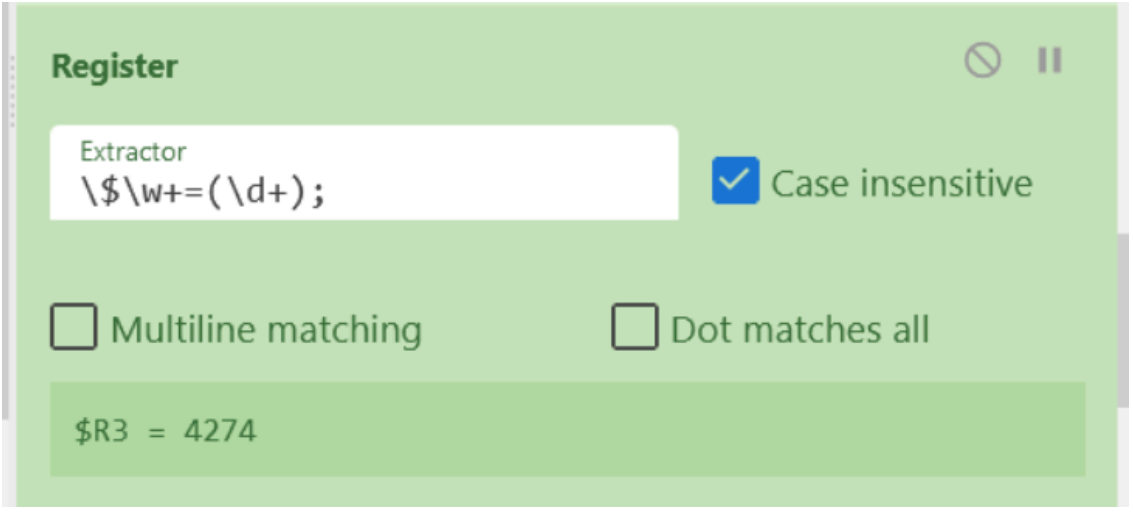
To Decode stage 3, we must perform the following actions

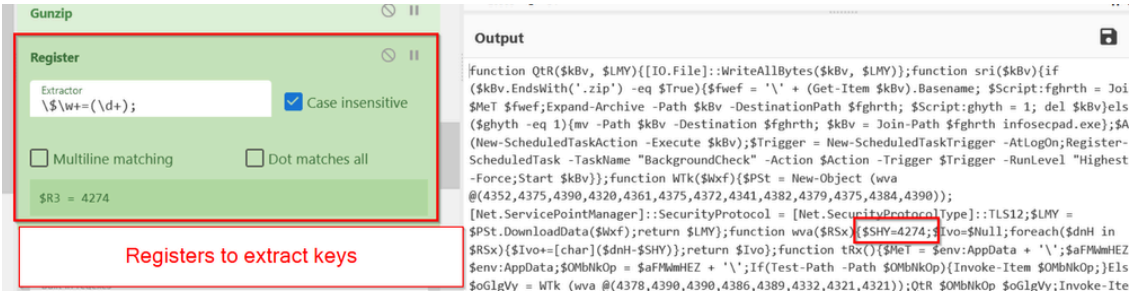
- Use Registers to Extract The Subtraction Value
- Use Regex to extract the decimal arrays
- Use Forking to Separate the arrays
- Use another Fork to Separate the individual decimal values
- Use a find/replace to append the subtraction value
- Perform the Subtraction
- Restore the text from the resulting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

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

Apply a capture group (round brackets) to the decimal component, as we want to store and use this later.



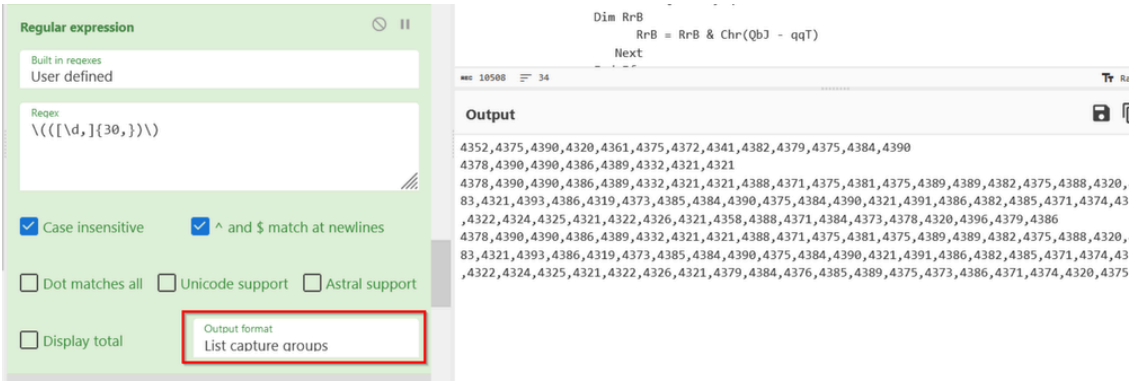


Operation 17 - Extracting and Isolating the Decimal Arrays

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

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 has been converted to a capture group by adding round brackets without escapes.

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

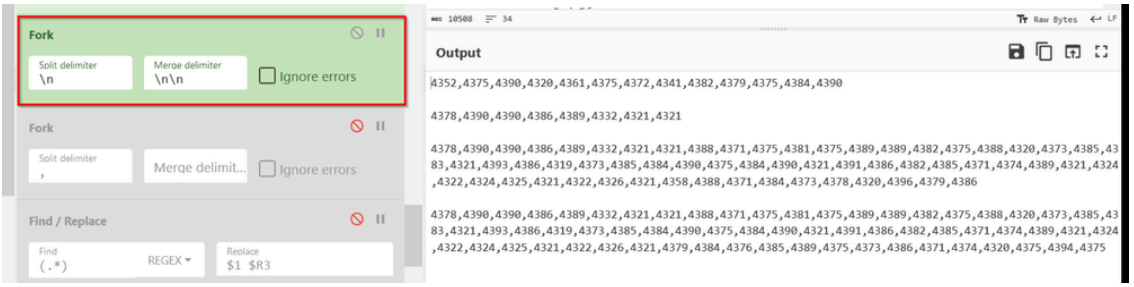


Operation 18 - Separating the Arrays With Forking

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

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

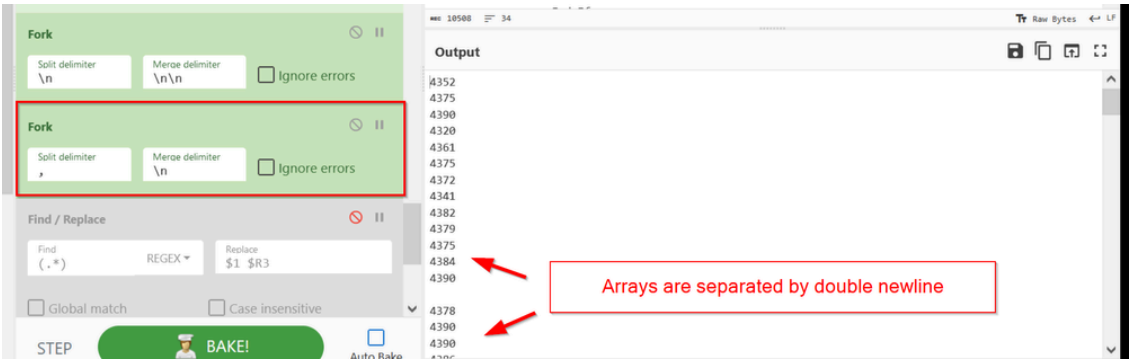
In the interests of readability, we can specify our merge delimiter as a double newline. The double newline does nothing except make the output easier to read.



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.

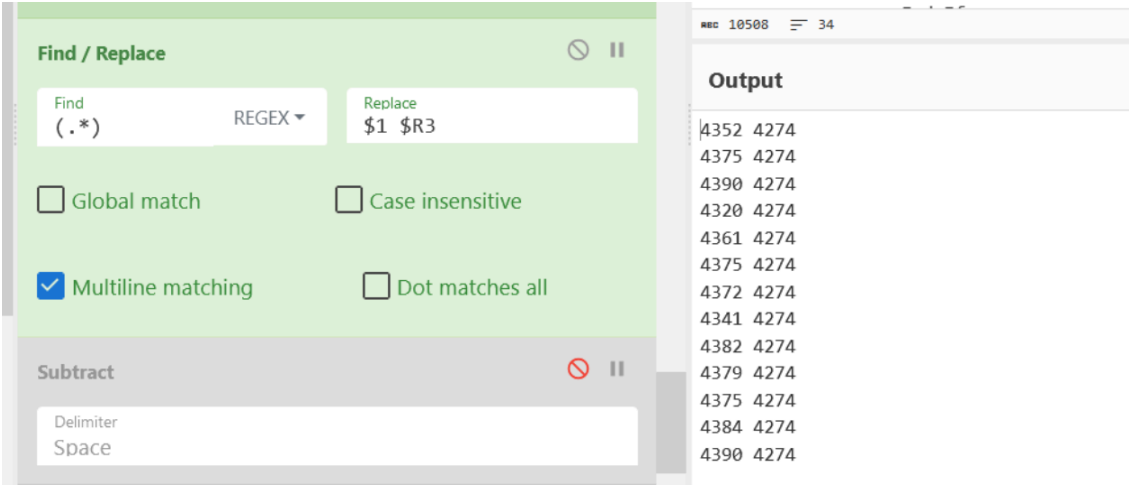
We can do this with another Fork operation, specifying a comma delimiter (as this is what separates our decimal values) and a merge delimiter of newline. Again, this new line does nothing but improve readability.



Operation 20 - Appending Subtraction Values

With the decimal values isolated, we can use a previous technique to capture each line and append the subtraction key currently stored in `$R3`.

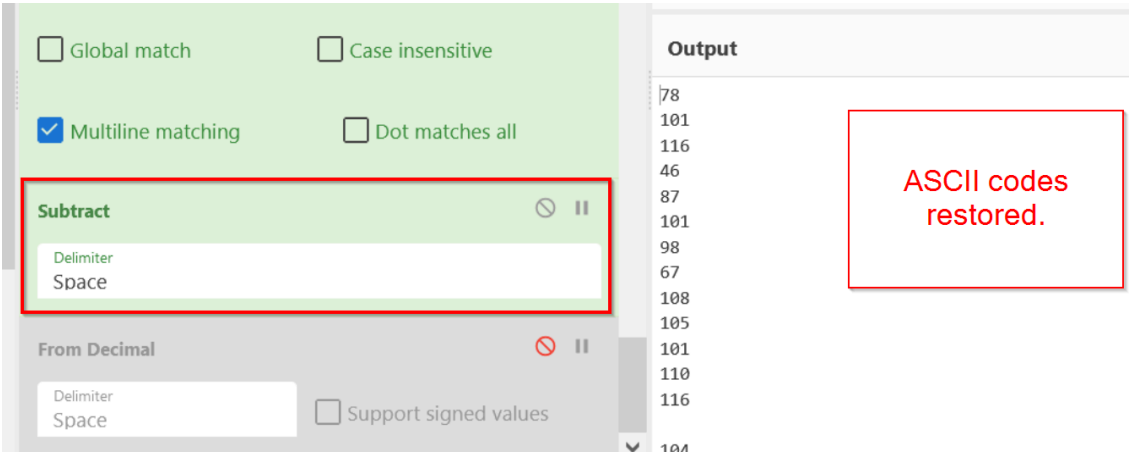
We can see the subtraction key appended to each line containing a decimal value.



Operation 21 - Applying the Subtraction Operation

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

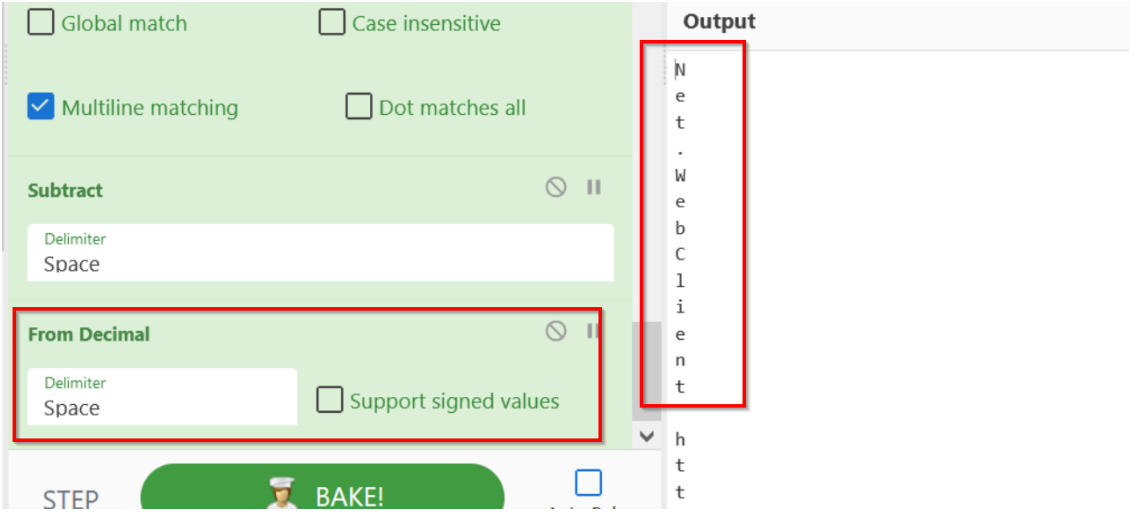
This restores the original ASCII char codes so we can decode them in the next step.



Operation 22 - Decoding the ASCII Codes

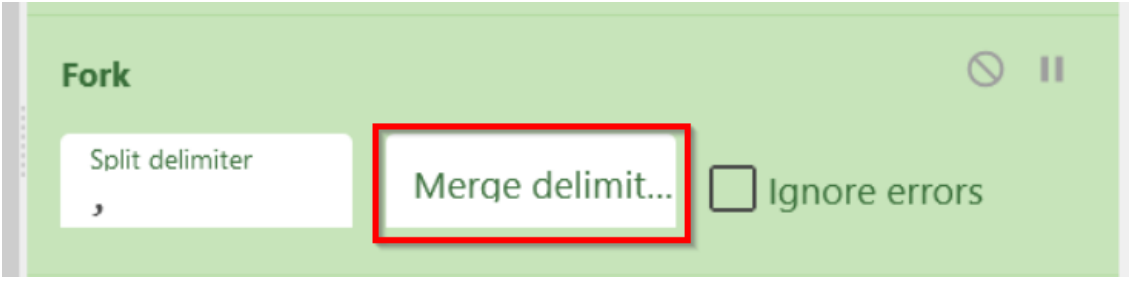
With the ASCII codes restored in their original decimal form, we can apply a from decimal operation to restore the original text.

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

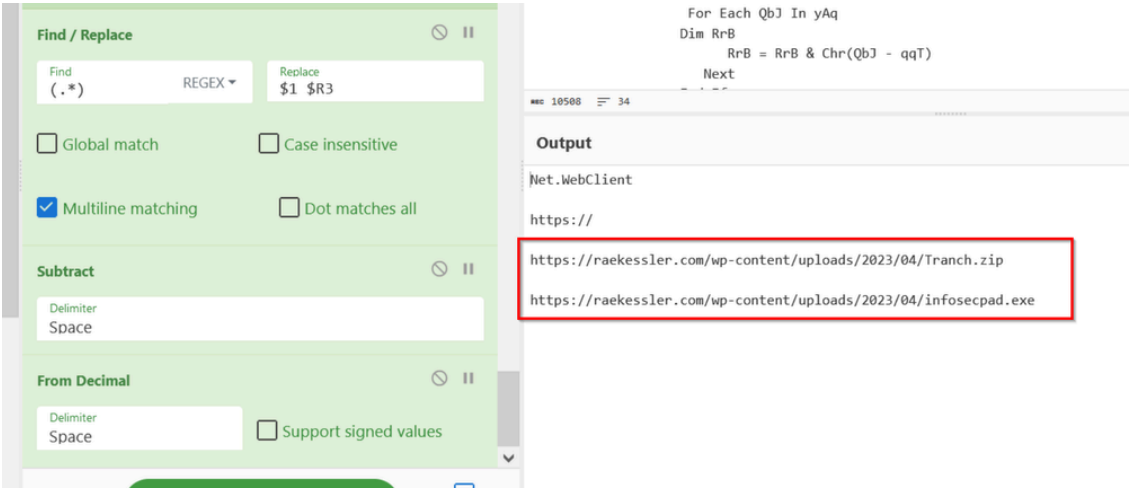


Now that the content is decoded, we can remove the readability step we added in Operation 19.

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



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



Video Walkthrough

Sign up for Embee Research

Malware Analysis, Detection Engineering and Threat Intelligence

No spam. Unsubscribe anytime.
